NAVAL POSTGRADUATE SCHOOL
Monterey, California

AD-A283 446

DTIC
SELECTED
AUG 1 9 1994
F
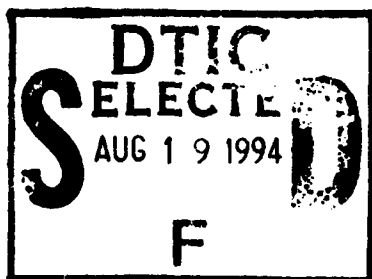
THESIS

ENHANCEMENT OF EMAG:
A 2-D ELECTROSTATIC AND MAGNETOSTATIC
SOLVER FOR MATLAB

by

David Patrick Wells

September, 1994

Thesis Advisor: Jovan E. Lebaric

Approved for public release; distribution is unlimited.

DTIC QUALITY INSPECTED 1

94-26392

'94 8 18 1 6 3

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE Sep 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE Enhancement of EMAG: A 2-D Electrostatic and Magnetostatic Solver for MATLAB		5. FUNDING NUMBERS
6. AUTHOR(S) David P. Wells		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE A

13. ABSTRACT *(maximum 200 words)*

    This thesis presents the theory and development involved in the enhancement of EMAG, a 2-D electrostatic and magnetostatic solver, to allow it to solve problems involving rotational symmetry. EMAG 2.0 solves rotationally symmetric problems using discrete forms of the Poisson equations for electrostatics and magnetostatics in cylindrical coordinates. EMAG 2.0 is written entirely in MATLAB script format. It allows users to define electrostatic or magnetostatic problems on a 2-D grid and solve the problem for the potentials at uniformly spaced nodes on the grid. Graphical displays allow the users to visualize contour or mesh plots of potential, vector plots of electric or magnetic fields and to calculate the charge or current enclosed in a user defined region of the grid.

    The EMAG 2.0 computational grid has a simulated open boundary which is generated by the Transparent Grid Termination (TGT) technique. This boundary is unique to the type of system being solved. This thesis presents and compares two different methods for generating this boundary, one involving a probabilistic model of the system and the other using a direct matrix solution approach. Optimization of the Transparent Grid Termination technique is also explored.

14. SUBJECT TERMS Electromagnetics, education, computer software.			15. NUMBER OF PAGES 136
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Enhancement of EMAG:
A 2-D Electrostatic and Magnetostatic Solver for MATLAB

by

David P. Wells
Captain, United States Marine Corps
B.S., United States Naval Academy, 1988

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1994

Author: _____
David P. Wells

Approved by: _____
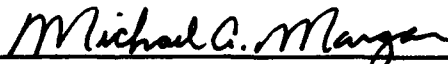Jovan E. Lebaric, Thesis Advisor

_____
David C. Jenn, Second Reader

_____
Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

ii

ABSTRACT

This thesis presents the theory and development involved in the enhancement of EMAG, a 2-D electrostatic and magnetostatic solver, to allow it to solve problems involving rotational symmetry. EMAG 2.0 solves rotationally symmetric problems using discrete forms of the Poisson equations for electrostatics and magnetostatics in cylindrical coordinates. EMAG 2.0 is written entirely in MATLAB script format. It allows users to define electrostatic or magnetostatic problems on a 2-D grid and solve the problem for the potentials at uniformly spaced nodes on the grid. Graphical displays allow the users to visualize contour or mesh plots of potential, vector plots of electric or magnetic fields and to calculate the charge or current enclosed in a user defined region of the grid.

The EMAG 2.0 computational grid has a simulated open boundary which is generated by the Transparent Grid Termination (TGT) technique. This boundary is unique to the type of system being solved. This thesis presents and compares two different methods for generating this boundary, one involving a probabilistic model of the system and the other using a direct matrix solution approach. Optimization of the Transparent Grid Termination technique is also explored.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Jovan E. Lebaric, for his tremendous assistance

and support during the course of my research. His guidance and friendship have made it

a truly enjoyable learning experience. I would like to thank my wife, Laura, and my

daughter, Erin, for the love and support which have motivated me throughout the course

of my graduate work.

Accesion For		
NTIS CRA&I	☑	
DTIC TAB	☐	
Unannounced	☐	
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

I. INTRODUCTION

A. OVERVIEW

This thesis will describe the enhancement of EMAG, a Finite Difference Electrostatic and Magnetostatic problem solver toolbox for MATLAB, to enable it to solve problems involving rotational symmetry. Further, it will detail the results of the associated research concerning the modeling of open boundary conditions. It will begin with an introduction to the original version of EMAG and the method used in modeling its boundary conditions, Transparent Grid Termination (TGT). In Chapter II, the finite difference equations used in solving rotationally symmetric problems will be developed. These results will be used in the third chapter as the solution to the open boundary problem for rotationally symmetric systems is explored. Chapter III will also present an analysis of two different methods used for solving the open boundary problem and will compare the two methods with respect to accuracy, speed of calculation, and computational memory requirements. Chapter IV will present examples of EMAG 2.0 capabilities and compare these results with known solutions.

B. EMAG

EMAG was originally developed by Roger Manke, Jr. at Rose-Hulman Institute of Technology. It is a MATLAB toolbox that solves user defined electrostatic and magnetostatic problems on a uniform square grid, subject to a distant Dirichlet boundary of zero potential. Potentials at equally spaced nodes within the grid are calculated using discretized forms of Poisson's equation. Using a mouse and keyboard, EMAG users define a problem by drawing media and sources on a 2-D computational grid. For electrostatic problems, the types of media include dielectric and perfect electric conductor (PEC) material. For problems in magnetostatics, the media is magnetic or perfect magnetic reluctor (PMR) material. PMR is a non-physical medium characterized by constant magnetic vector potential throughout its volume and therefore, is the dual of PEC. PMR media has infinite reluctivity or zero permeability. Two computational grid sizes are available to the user. The 17x17, "coarse" computational grid provides rapid results with one third of the resolution of the "fine" grid. The 51x51, "fine" computational grid provides greater fidelity but requires more computation time. EMAG output is in the form of graphical displays and numerical data. EMAG graphical displays include 3-D mesh plots of potential, equipotential contour plots, and plots of electric or magnetic fields. EMAG can also calculate numerical results for the enclosed charge within a user specified area on an electric field plot or enclosed current on a magnetic field plot. Furthermore, all output parameters (such as the matrix containing the

2

calculated nodal potentials) are available to the user for analysis. The original version of EMAG solved problems that were invariant along an infinite axis into and out of the computational grid (z-invariant). In Reference 1, Manke developed the finite difference (FD) equations used in EMAG for solving z-invariant problems. Chapter II of this thesis will present the development of the equations used in EMAG 2.0 for systems involving rotational symmetry about a central axis.

The EMAG computational grid boundary is a layer of nodes which simulate a distant, homogeneous Dirichlet boundary of zero potential. This boundary is developed using a method referred to as Transparent Grid Termination (TGT). Use of the distant Dirichlet boundary, in effect, surrounds the EMAG computational grid with free space and maximizes EMAG's accuracy. TGT is used to allow the lengthy process of boundary calculations to be performed only once. These results are then stored as a file and used by EMAG in the solution of its open boundary problems. The theory behind TGT and the methods used in calculating the boundary conditions for rotationally symmetric systems will be explored in detail in Chapter III. [Ref 1.]

II. FD SOLUTION OF ROTATIONALLY SYMMETRIC GEOMETRIES

A. DISCRETIZED POISSON EQUATION

This section will detail the process of developing the discretized Poisson equations for rotationally symmetric systems. Although the electrostatic and magnetostatic Poisson equations are dual equations, for rotationally symmetric systems their discretized forms are quite different from one another. This is unlike the discretized equations for z-invariant systems which are identical for both electrostatics and magnetostatics. Development of the discretized electrostatic Poisson equation begins by considering an elementary volume in cylindrical coordinates as shown in Figures 1 and 2. Figure 1 shows an elementary volume in cylindrical coordinates and the locations of neighboring discrete nodes relative to that volume. Figure 2 depicts a cross section of that volume.

4
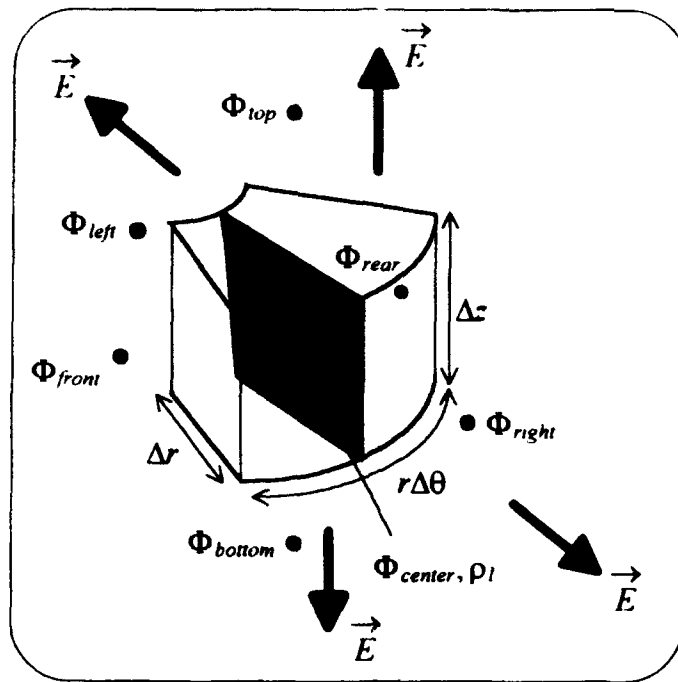
Figure 1. Elementary Volume
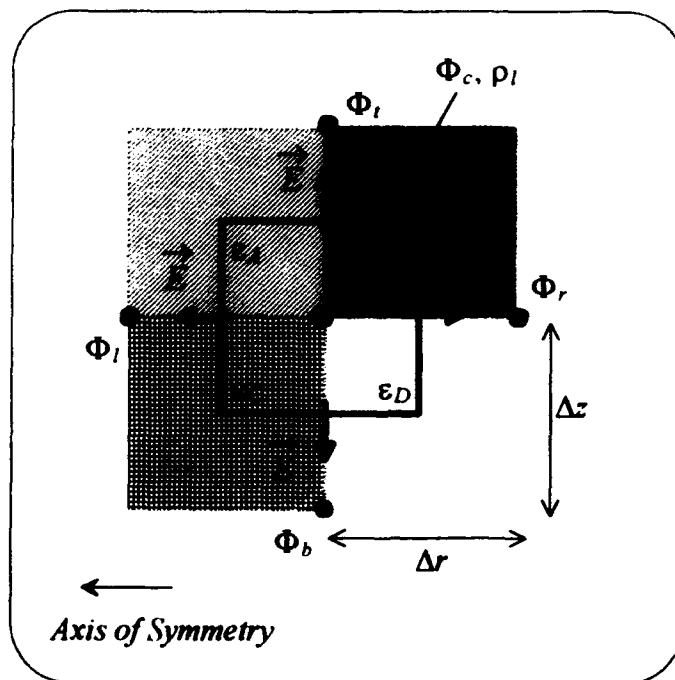
Figure 2. Elementary Volume Cross Section

5

Poisson's equation relates electric scalar potential to the enclosed charge distribution and media by

$$\nabla^2 \Phi = -\frac{\rho}{\varepsilon} . \tag{1}$$

The goal of this section is to develop an equation which relates the potential at a given node (say the center node) to the potentials of its neighboring nodes, the charge within the volume, and the media parameters. Since this system is rotationally symmetric, there is no variation in the media or the sources with respect to the angle of rotation and $\Phi_{front} = \Phi_{rear}$. The center node is separated from its neighbors by four annular regions of media, each of dimension Δr by Δz in cross section. Since symmetry requires that potential remains constant with respect to θ, the θ component of the electric field is zero. Using the integral form of Gauss' Law

$$\oint_S \vec{D} \cdot d\vec{s} = Q_{enclosed} , \tag{2}$$

the constitutive relationship

$$\vec{D} = \varepsilon \vec{E} , \tag{3}$$

6

and assuming that the fields are constant over each face of the elementary volume, the charge inside the elementary volume can be related to the electric flux penetrating outward through each of the remaining four sides of the volume by

$$-E_r(r-\frac{\Delta r}{2},z)\cdot[(r-\frac{\Delta r}{2})\Delta\theta\frac{\Delta z}{2}\cdot\varepsilon_A + (r-\frac{\Delta r}{2})\Delta\theta\frac{\Delta z}{2}\cdot\varepsilon_C \quad Left \quad (4)$$

$$+E_r(r+\frac{\Delta r}{2},z)\cdot[(r+\frac{\Delta r}{2})\Delta\theta\frac{\Delta z}{2}\cdot\varepsilon_B + (r+\frac{\Delta r}{2})\Delta\theta\frac{\Delta z}{2}\cdot\varepsilon_D \quad Right$$

$$+E_z(r,z+\frac{\Delta z}{2})[(r+\frac{\Delta r}{4})\Delta\theta\frac{\Delta r}{2}\cdot\varepsilon_A + (r+\frac{\Delta r}{4})\Delta\theta\frac{\Delta r}{2}\cdot\varepsilon_B \quad Top$$

$$-E_z(r,z-\frac{\Delta z}{2})\cdot[(r-\frac{\Delta r}{4})\Delta\theta\frac{\Delta r}{2}\cdot\varepsilon_C + (r+\frac{\Delta r}{4})\Delta\theta\frac{\Delta}{2}\cdot\varepsilon_D \quad Bottom$$

$$= Q_{enclosed}$$

where $Q_{enclosed}$ is related to the volume charge density by

$$Q_{enclosed} = \rho_v\, r\,\Delta\theta\,\Delta z\,. \qquad (5)$$

The next step is to relate the electric field components to the potentials of the discrete nodes. Using the definition of electric scalar potential

$$\vec{E} = -\nabla\Phi\,, \qquad (6)$$

7

leads to the following FD equations:

$$E_r(r + \frac{\Delta r}{2}, z) = \frac{\Phi_r - \Phi_c}{\Delta r} \qquad (7)$$

$$E_r(r - \frac{\Delta r}{2}, z) = \frac{\Phi_c - \Phi_l}{\Delta r} \qquad (8)$$

$$E_z(r, z + \frac{\Delta z}{2}) = \frac{\Phi_t - \Phi_c}{\Delta z} \qquad (9)$$

$$E_z(r, z - \frac{\Delta z}{2}) = \frac{\Phi_c - \Phi_b}{\Delta z} . \qquad (10)$$

Substituting equations (7) through (10) into equation (4) and selecting $\Delta z = \Delta r = \Delta l$ to establish a uniform grid yields

$$(1 - \frac{\Delta r}{2r})(\frac{\varepsilon_A + \varepsilon_C}{2}) \Phi_l + (1 + \frac{\Delta r}{2r})(\frac{\varepsilon_B + \varepsilon_D}{2})\Phi_r \qquad (11)$$

$$+ [\frac{(1 - \frac{\Delta r}{4r})\varepsilon_A + (1 + \frac{\Delta r}{4r})\varepsilon_B}{2}] \Phi_t + [\frac{(1 - \frac{\Delta r}{4r})\varepsilon_C + (1 + \frac{\Delta r}{4r})\varepsilon_D}{2}] \Phi_b$$

$$- [(2 - \frac{3\Delta r}{4r})(\frac{\varepsilon_A + \varepsilon_C}{2}) + (2 + \frac{3\Delta r}{4r})(\frac{\varepsilon_B + \varepsilon_D}{2})] \Phi_c = -\rho_v(\Delta l)^2 = -\rho_l \quad (C/m)$$

which is the discretized Poisson equation for a rotationally symmetric electrostatic system.

8

The development of the magnetostatic Poisson equation also begins with the
elementary volume in cylindrical coordinates. A cross section of the elementary volume is
shown in Figure 3.
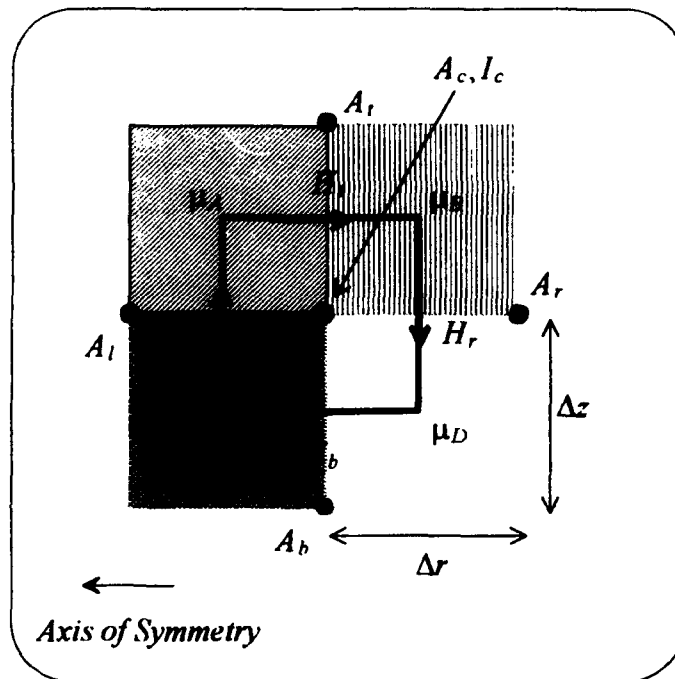
Figure 3. Magnetostatic Elementary Volume

In this figure, the current (I) and the magnetic vector potentials (A_t, A_r, A_l, A_b and A_c)
are all in the θ direction. This allows the magnetic vector potential to be treated as a
scalar. Using this fact along with the integral form of Ampere's Law, a discretized form of
Ampere's Law can be developed. Ampere's Law relates the magnetic field around a

9

closed path to the total current flowing through any surface bounded by that path and is given by

$$\oint_C \vec{H} \cdot \vec{dl} = \int_S \vec{J} \cdot \vec{ds} \qquad (12)$$

where H is the magnetic field and J is the current density. Selecting a uniform grid for EMAG ($\Delta l = \Delta r = \Delta z$) results in a discretized form of Ampere's Law

$$H_l \Delta r + H_r \Delta r + H_t \Delta r + H_b \Delta r = J_c(\Delta r)^2 = I_c \ . \qquad (13)$$

The next step is to solve for the magnetic fields along each side of the elementary volume containing the center node. This is accomplished by using the relationship

$$\vec{H} = \frac{1}{\mu}(\nabla \times \vec{A}) \qquad (14)$$

which relates the magnetic field to the magnetic flux density (the curl of the magnetic vector potential) and the permeability of the media. Assuming that the magnetic flux is constant along each side of the contour surrounding the elementary volume, the reluctivity, $1/\mu$, will be the average of the reluctivities of the two homogeneous annular regions through which the flux penetrates.

10

Using this value for reluctivity along with the definition of curl in cylindrical coordinate systems and the fact that all of the magnetic vector potentials are exclusively in the θ direction, equation (14) becomes

$$\vec{H} = (\frac{1}{\mu})_{avg} [-\frac{\partial A_\theta}{\partial z} \vec{r} + \frac{1}{r}\frac{\partial(rA_\theta)}{\partial r} \vec{z}]$$ (15)

where r is the center node's radial distance from the axis of rotation. Applying equation (15) to all four sides of the volume results in the following four FD equations for the magnetic fields shown in Figure 3:

$$H_t = \frac{1}{2}(\frac{1}{\mu_A} + \frac{1}{\mu_B})(\frac{A_c - A_t}{\Delta z})$$ (16)

$$H_b = \frac{1}{2}(\frac{1}{\mu_C} + \frac{1}{\mu_D})(\frac{A_c - A_b}{\Delta z})$$ (17)

$$H_r = \frac{1}{2}(\frac{1}{\mu_B} + \frac{1}{\mu_D})(\frac{rA_c - (r + \Delta r)A_r}{\Delta r})(\frac{1}{r + \frac{\Delta r}{2}})$$ (18)

$$H_l = \frac{1}{2}(\frac{1}{\mu_A} + \frac{1}{\mu_C})(\frac{rA_c - (r - \Delta r)A_l}{\Delta r})(\frac{1}{r - \frac{\Delta r}{2}})$$ (19)

11

Substituting equations (16) through (19) into equation (13) yields

$$[(\frac{1}{\mu_A} + \frac{1}{\mu_C})(1 + \frac{1}{2}(\frac{1}{\frac{2r}{\Delta r} - 1})) + (\frac{1}{\mu_B} + \frac{1}{\mu_D})(1 - \frac{1}{2}(\frac{1}{\frac{2r}{\Delta r} + 1}))] A_c \qquad (20)$$

$$- [\frac{1}{2}(\frac{1}{\mu_A} + \frac{1}{\mu_C})](1 - \frac{1}{\frac{2r}{\Delta r} - 1}) A_l - [\frac{1}{2}(\frac{1}{\mu_B} + \frac{1}{\mu_D})](1 + \frac{1}{\frac{2r}{\Delta r} + 1}) A_r$$

$$- [\frac{1}{2}(\frac{1}{\mu_A} + \frac{1}{\mu_B})] A_t - [\frac{1}{2}(\frac{1}{\mu_C} + \frac{1}{\mu_D})] A_b = I_c \qquad \text{(Amps)}.$$

Equation (20) is the discretized magnetostatic Poisson equation which relates the

potential of the center node to the potential of its four nearest neighboring nodes and the

current through the center node. Although equations (11) and (20) look very different

from one another, they both tend toward the z-invariant Poisson equations used in

EMAG as r tends toward infinity. This fact is the basis for the approach used to modify

the EMAG code presented in the next section.

12

B. IMPLEMENTATION OF DISCRETE POISSON'S EQUATION IN EMAG 2.0

In order to solve rotationally symmetric problems, the original version of EMAG needed to be modified in three ways. First, in the equation solver subprograms, code needed to be added which applied the Poisson equations derived in the previous section. These types of modifications are the subject of this section. The second set of modifications, to be discussed in Chapter III, involved the use of rotational symmetry equations to develop the TGT boundary data. The third set of modifications involved addition of code to allow EMAG 2.0 users to choose between z-invariant and rotationally symmetric systems. This last set if modifications will not be discussed except in the context of how EMAG 2.0 is used. All modified EMAG subprograms can be found in Appendix A.

The original version of EMAG uses two different methods for calculating the potentials across the computational grid. The first method, intended for solving coarse (17 x 17) grid problems, utilizes a system matrix containing all information about media and the TGT boundary. This system matrix is generated by the MATLAB script file *makesys2.m*. The script file *matsolve.m* then applies the user specified source information and solves the system of equations using matrix operations. To modify the coarse solver for rotational symmetry, equations (11) and (20) were used. Specifically, the factors in these equations which weight the media parameters based on the center node's relative distance from the axis of symmetry (such as $(1 - \frac{\Delta r}{2r})$) were inserted into

13

duplicates of the z-invariant equations. These equations are used instead of the z-invariant set when the user specifies that the problem to be solved has rotational symmetry. The system matrix approach was not used in the original version of EMAG in solving fine (51 x 51) grid problems, because of the large size of a fine grid system matrix and the fact that the sparse matrix tools were not yet included in MATLAB. For a fine grid of this size, the system matrix would contain nearly 6.8 million elements and require 54 megabytes of memory if stored as a full matrix [Ref. 1]. As a result, a second, more memory conservative solution approach was utilized.

The method originally used in EMAG for solving fine grid problems was a Jacobi iterative solver. (The reasons for choosing a Jacobi solver are detailed in Reference 1.) This solver is contained in the MATLAB script file *itersoln.m*. The Jacobi solver can utilize saved results from previous problems, the results of the coarse solver or a default set of potentials (zero except where known *a priori*) as the starting point for its iterations. The EMAG user specifies a desired percent error at which to stop iterating the solution. The computational time required to solve a problem using this approach increases with the degree of accuracy required. Because the modifications leading to EMAG 2.0 were accomplished using MATLAB 4.0, sparse matrix operations were utilized in the system matrix solver to allow it to be used for both coarse and fine grid solutions. The Jacobi solver however, was not eliminated since it has the useful advantage that it can use previous solutions as the starting point for its iterations, thereby increasing its speed of

convergence to the solution. For example, by using the coarse grid solution as a starting point for a fine grid calculation, the iterative solver can often run faster than the system matrix solver with an estimated error less than 1%. The system matrix approach, which requires the same computational time regardless of the problem, cannot take advantage of a previous solution, but generally runs faster than the Jacobi solver without an accurate starting set of potentials. Accordingly, in EMAG 2.0 the user has a choice of using either the Jacobi solver or the system matrix solver when solving a problem on the fine grid. Fine grid problems using the system matrix approach are solved using the *matsolvf.m* and *makesysf.m* script files. The iterative solver in *itersoln.m* has been modified to include the Poisson equations for rotational symmetry. Modifications to *itersoln.m* were similar to those made to *makesys.m* in that the rotational symmetry unique terms were applied as factors weighting the media parameter terms.

When EMAG 2.0 users desire to solve a rotationally symmetric problem, they simply answer "y" (or "Y") to the question "Do you want to solve a rotationally symmetric system?" in the "New Domain Region" selection part of the EMAG session. Once this choice is made, EMAG 2.0 remains in the rotational symmetry mode until a new domain region is requested. As a reminder that EMAG is in the rotational symmetry mode, a border around the computational grid is plotted in red. This border is white in the z-invariant mode. The user then defines cells of media and sources on the computational grid in the same manner as in the z-invariant case except that only the right half of the

15

cross section of the system is visible. For problems in electrostatics, the axis of symmetry is parallel and one half of an inter-nodal distance (Δl) to the left of the left side of the computational grid. For magnetostatic problems, the axis of symmetry is parallel and a whole inter-nodal distance to the left. These arrangements take advantage of symmetry, maximize the useful input area of the computational grid, and eliminate the mirror images that would result if the axes of symmetry had been chosen to be in the center of the computational grid. They also allow reuse of the drawing routines used for z-invariant systems. A sample electrostatic problem on the EMAG 2.0 rotational symmetry computational grid is shown in Figure 4.
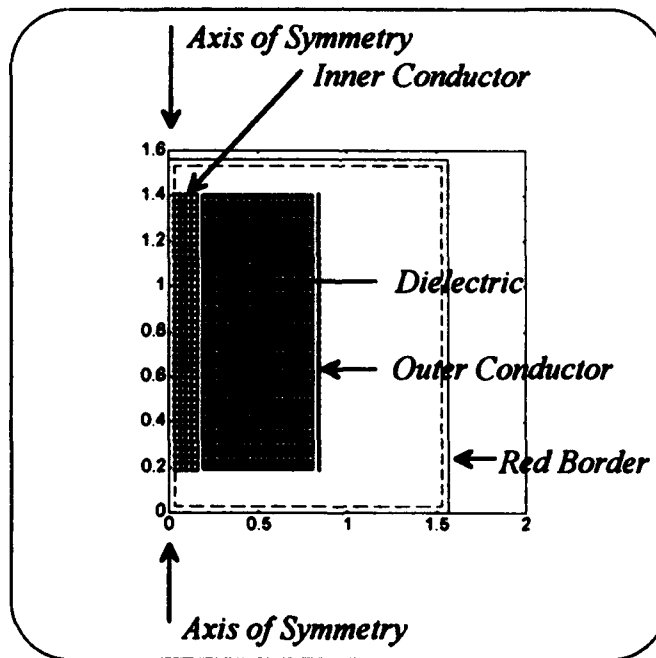
Figure 4. Section of Coaxial Cable on Computational Grid

Figure 4 depicts the right half cross section of a short segment of a coaxial cable as it would appear on the EMAG 2.0 computational grid. (If the user had not requested a rotationally symmetric problem, this same picture would represent a parallel plate capacitor extending infinitely into and out of the computational grid along the z-axis.)

In addition to the changes to the system solver programs listed above, EMAG's subroutines required several other modifications to accommodate rotational symmetry. First, the subroutines *cefield.m* and *fefield.m*, which plot the electric or magnetic fields for the coarse and fine grids respectively, were modified so that the magnetic field would be calculated using the definition of curl in cylindrical coordinates. Next, the enclosed charge calculation subprogram, *q_calc.m*, which performs a flux integral, was modified to account for the fact that the area of the faces of the elementary volume are functions of the volume's distance from the axis of rotation. This modification results in the charge enclosed in a user specified area of the rotationally symmetric computational grid being expressed as total charge (in Coulombs) instead of charge per unit length (Coulombs/meter) as it is in the z-invariant case. The last modification was applied to the enclosed current subroutine *i_calc.m*. This subroutine was modified so that the enclosed current for rotationally symmetric systems is calculated using the definition of curl in cylindrical coordinates. The modified versions of all the above subroutines are included in Appendix A.

17

III. MODELING OF OPEN BOUNDARY

A. THEORY OF TRANSPARENT GRID TERMINATION (TGT)

In order to quickly and accurately solve open boundary problems, EMAG uses a technique called Transparent Grid Termination (TGT) to simulate the existence of a zero potential, Dirichlet boundary far away from the user defined problem on the computational grid. Using the known zero potential on the Dirichlet boundary and defining all space outside EMAG's computational grid to be source free and homogeneous, a very large system of Poisson equations involving the nodes inside the Dirichlet boundary is *partially* solved in advance. The end result of this process is a system of equations which relate the potentials of the nodes on the first layer outside the computational grid (called the TGT boundary) to all of the nodes just inside of this layer, given that there exists a distant boundary of zero potential. These equations are stored in the form of a matrix (called the TGT matrix) which, once calculated, replaces the many concentric layers of nodes in the homogeneous, source free region between the computational grid and the distant Dirichlet boundary.

Since this original "buffer" zone of nodes was defined to be homogeneous and source free, this TGT matrix *does not change* from problem to problem and in no way depends on what may be defined within the computational grid for any particular problem. As such, the TGT matrix is stored and reused over and over to solve any user defined

problem quickly. Since the TGT boundary relationships are calculated by partially

solving the larger physical system with a Dirichlet boundary, TGT provides exactly the

same solution as would be obtained by solving the user defined problem with a distant

homogeneous Dirichlet boundary. However "solving through" this unchanging buffer

zone once and storing the results eliminates repetitive and time consuming computations.

Solving the whole open boundary problem would require several hours on even the

fastest PC. The vast majority of this time would be spent solving through the buffer zone

which does not change from problem to problem. TGT simply allows one to devote this

time only once in the calculation of the TGT matrix and then to use the results of this

partially solved system for different problems. [Ref. 1]

Figures 5 and 6 illustrate the relationships between the computational grid and the

Dirichlet boundary for z-invariant and rotationally symmetric systems respectively.

The arrows in these two figures show how each node on the TGT boundary is globally

related to every node on the outer layer of the computational grid and the Dirichlet

boundary while each node on the computational grid is related only locally to its four

nearest neighbors [Ref. 2]. Notice that unlike the z-invariant computational grid

(Figure 5), the rotationally symmetric computational grid (Figure 6) has a distant

Dirichlet boundary on only three sides. The left side of the rotationally symmetric

computational grid for electrostatic systems is adjacent to the axis of symmetry and as a

result, has a Neuman boundary of zero gradient to its left. The axis of symmetry for

19
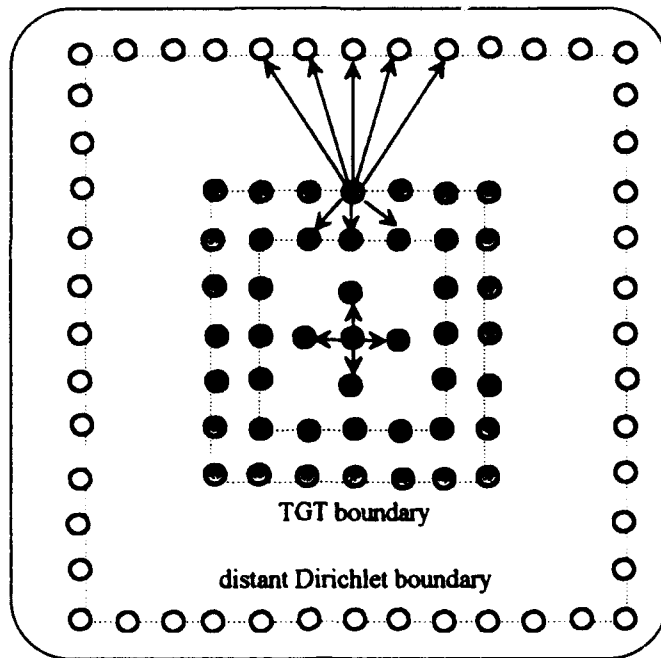
Figure 5. z - Invariant System
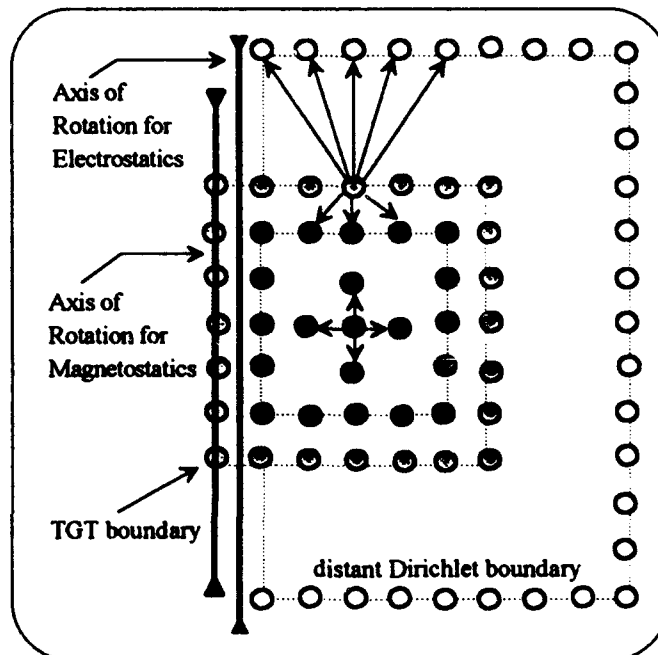
Figure 6. Rotationally Symmetric System

20

the magnetostatic rotationally symmetric system is placed on the left side of the TGT

boundary and is a homogeneous Dirichlet boundary. The reason for the different

placements of the rotationally symmetric electrostatic and magnetostatic computational

grids is that for magnetostatic systems with only θ directed components of magnetic

vector potential, the potential is known to be zero on the axis of rotation. In contrast, for

the electrostatic system, it is the gradient of the electric scalar potential that is zero on the

axis of rotation. Different grid placements, therefore, take advantage of these known

facts in establishing the boundary conditions. As shown in Figures 5 and 6, EMAG

always uses a square computational grid and the TGT boundary is a layer of nodes one

inter-nodal distance outward from the last layer of the computational grid. This TGT

boundary contains all information about the distant Dirichlet boundary required to solve

any problem drawn by a user on the computational grid with the same accuracy as

solving the much larger problem within the distant Dirichlet boundary. With the TGT

boundary matrix calculated in advance, EMAG only needs to solve the sparse system of

equations depicted by the topology map of Figure 7. [Ref. 2]

Topology Map

Spiral Node Labelling
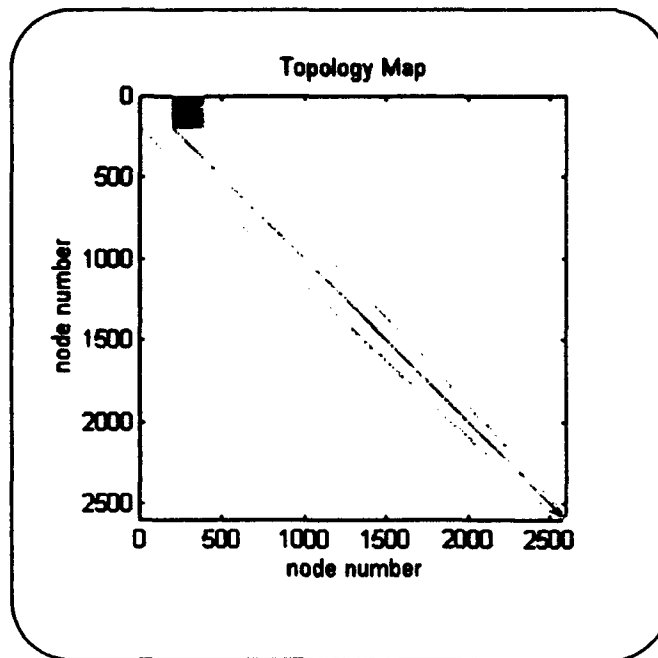
Figure 7. EMAG System Sparsity Pattern

Figure 7 is a mapping of the non-zero elements in the matrix which relates the nodes

on the computational grid and the TGT boundary to one another. A spiral node labeling

pattern is used, starting from the upper left node on the TGT boundary and spiraling

clockwise inward to the center of the computational grid. The main "diagonal" is three

elements wide and represents the relationship between a node and its nearest neighboring

nodes on its right and left. These relationships are derived using the discretized Poisson's

equations, equations (11) or (20). The upper diagonal represents the terms which relate

each node to its neighboring node immediately inward, toward the center. The lower

22

diagonal represents the terms which relate each node to its neighbor outward, away from the center. The dense square area in the upper left corner of the map represents the TGT matrix. It is this densely packed matrix which relates the potentials of the nodes on the TGT boundary to the nodes on the outer edge of the computational grid given the fact that there exists a distant Dirichlet boundary of zero potential. Whereas the diagonals represent local relationships between a node and its neighbors, the TGT matrix relates each node on the TGT boundary globally to all the nodes on the outer edge of the computational grid. The spiral numbering scheme is used in order that the TGT matrix be densely packed with no non-zero elements, thereby minimizing its size. The next two sections of this chapter will present two different approaches to calculating the TGT matrix. The final section of this chapter will compare these techniques and present the results of research into the optimization of TGT. [Ref. 2]

B. CALCULATING THE TGT MATRIX: THE MATRIX SOLUTION METHOD

The starting point for calculating the TGT matrix of coefficients is always the discretized Poisson equation for the given system. Although this chapter will describe the process of calculating the TGT coefficients for a rotationally symmetric system, the procedure is "generic" and has also been used to calculate TGT coefficients for z-invariant systems. Since the buffer zone between the distant Dirichlet boundary is homogeneous and source-free, the discretized Poisson equation for electrostatics (equation (11)) simplifies to a discretized Laplace equation given by

$$4\Phi_c = (1 - \frac{1}{2(\frac{r}{\Delta r})})\Phi_l + (1 + \frac{1}{2(\frac{r}{\Delta r})})\Phi_r + \Phi_t + \Phi_b . \qquad (21)$$

The corresponding equation for magnetostatics is

$$[4 - \frac{1}{2(\frac{r}{\Delta r} + \frac{1}{2})} + \frac{1}{2(\frac{r}{\Delta r} - \frac{1}{2})}]A_c = \qquad (22)$$

$$(1 - \frac{1}{2(\frac{r}{\Delta r} - \frac{1}{2})})A_l + (1 + \frac{1}{2(\frac{r}{\Delta r} + \frac{1}{2})})A_r + A_t + A_b .$$

Using the appropriate Laplace equation, it is a straightforward process to generate a set of three matrices which relate each node to its four neighbors. One matrix, M_l, relates the nodes on a layer (say layer m) in the homogeneous, source-free buffer zone to

their neighbors on the next layer of nodes outward toward the Dirichlet boundary (say layer l). The second matrix, M_n, relates nodes on layer m to the nodes on the next layer inward toward the computational grid (layer n). The third matrix, M_m, relates the nodes on a layer to themselves and their two neighbors on the same layer. Because these matrices have a discernible pattern, and are functions of the computational grid size and the relative size of the m-th layer of nodes, they can be generated algorithmically. Functions for creating these matrices via MATLAB are included in Appendix B. The three functions *makemlc.m*, *makemnc.m* and *makemmc.m* produce the M_l, M_n and M_m matrices respectively, for rotationally symmetric electrostatic systems. The functions *magmlc.m*, *magmnc.m* and *magmmc.m* perform the same functions for magnetostatic rotationally symmetric systems. The two input parameters for these functions are the number of nodes on a side of the desired square computational grid and the number of nodes on the right or left side of the m-th layer of nodes.

As shown in Figure 6, layers of nodes outside the computational grid are defined to have only three sides for a rotationally symmetric system. Symmetry makes it unnecessary to calculate the TGT coefficients for the left side of the TGT boundary. For electrostatic systems, these nodes are known to be at the same potential as their "mirror images" across the axis of rotation (Neuman boundary condition). The upper left and lower left nodes on the TGT boundary have the same TGT coefficients as their image nodes across the axis of rotation. The remaining nodes on the left side of the TGT

25

boundary have the same potential as their images and therefore have a single TGT

coefficient equal to one, which relates them only to their image node on the

computational grid. Figure 8 illustrates these relationships. In this figure, the gray nodes

are TGT boundary nodes and the black ones belong to the computational grid.
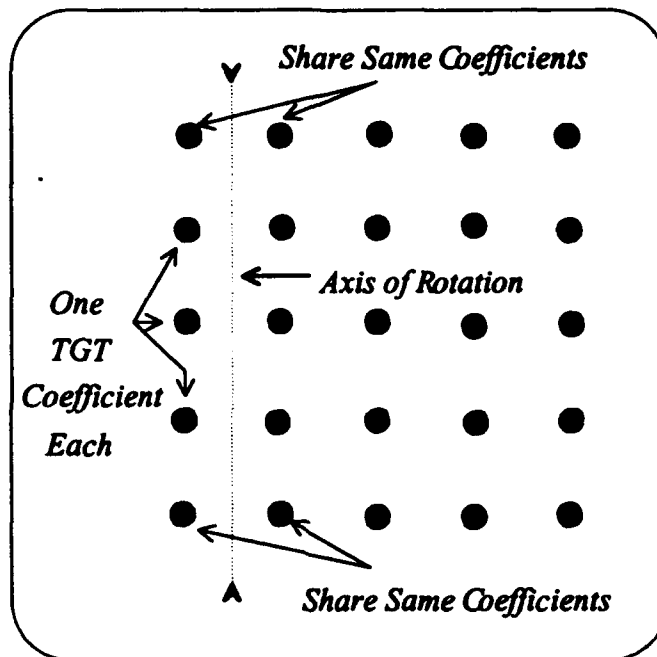
Figure 8. Left Side of TGT Boundary for Electrostatics

For magnetostatic systems, the potentials on the axis of rotation are known to be zero so

the left side of TGT boundary has been placed on the axis and all of the corresponding

coefficients have been set to zero (local Dirichlet boundary condition).

Samples of the output of each of these functions and their MATLAB script file listings are in Appendix B. The samples were generated for the first layer of nodes outside a three by three node computational grid such as shown in Figure 8. Output from these functions are m by x matrices ($x=l,m$ or n) for which the rows represent the nodes on the m-th layer and are numbered in a clockwise fashion starting with the node in the upper left corner of the layer. The columns represent the nodes on the l, m or nth layer and are numbered in the same manner. As can be seen from the sample outputs for this small computational grid, the matrices can get to be quite large. Fortunately, they are always very sparse. Also included in Appendix B are the three functions *makeml.m*, *makemn.m* and *makemm.m* which generate the M_l, M_n and M_m matrices for z-invariant systems along with samples of their output. Note that for z-invariant systems, these matrices are not functions of the inner grid size. The reason for this is that the layers of nodes outside the computational grid are concentric squares for this coordinate system. For rotationally symmetric systems, these layers of nodes are rectangular in shape and missing a left side. Further, it is important to note that there is no need to have different sets of programs to generate TGT coefficients for z-invariant electrostatic and magnetostatic systems since the z-invariant Laplace equations for electrostatics and magnetostatics are identical.

The process of solving for the TGT coefficients begins with a vector, $|a>$, which represents the potentials of the nodes on the Dirichlet boundary (layer a) and the node relationship matrices discussed above for the next layer inward (layer b).

27

The discretized Laplace's equation in matrix form is

$$B_b |b> \; = B_a |a> +B_c |c> \tag{23}$$

where $|b>$ and $|c>$ represent the potentials on the next two layers inward, b and c respectively. Continuing one layer inward leads to

$$C_c |c> \; = C_b |b> +C_d |d> \; . \tag{24}$$

Given that $|a>=0$ for a layer with zero potential, equations (23) and (24) can be combined and solved to find $|c>$, the potentials of the nodes on layer c, in terms of the nodes on the next layer inward. This solution is given by

$$|c> \; = (C_c - C_b (B_b)^{-1} B_c)^{-1} C_d |d> \; . \tag{25}$$

Since this solution does not depend on the potentials of nodes on layers a or b, these layers have been effectively eliminated from the problem. To calculate the TGT coefficients, this process of layer elimination is continued inward towards the computational grid. Since the subsequent expressions grow in length with every elimination, a special "termination" matrix can be defined for each node layer.

To simplify notation, define

$$\Gamma_b = B_b \qquad (26)$$

and

$$\Gamma_c = C_c - C_b (\Gamma_b)^{-1} B_c \qquad (27)$$

as the termination matrices for layers b and c respectively. [Ref. 2]

In general, the expression which relates the nodes on a layer to the nodes on the next layer inward is given by

$$|m> \ = (\Gamma_m)^{-1} M_n \ |n> \qquad (28)$$

where the generic termination matrix, Γ_m, can be calculated *iteratively* by

$$\Gamma_m = M_m - M_l (\Gamma_l)^{-1} L_m \quad . \qquad (29)$$

When this elimination process is continued inward until the TGT boundary layer is reached, the TGT coefficients are obtained by

$$TGT = (\Gamma_m)^{-1} M_n \quad . \qquad (30)$$

29

The MATLAB script files, *coefgenc.m* and *coefmagc.m*, used in calculating the TGT coefficients for rotationally symmetric electrostatic and magnetostatic systems respectively, can be found in Appendix B. These programs calculate the TGT coefficients for three sides of the TGT boundary using the algorithm discussed above. They then add the remaining terms for the fourth side based on symmetry and either known potentials or known gradients of potential. The last program contained in Appendix B is the TGT coefficient generating script file for z-invariant systems, *coefgen.m*. [Ref. 2]

C. CALCULATING THE TGT MATRIX: THE MONTE CARLO METHOD

The *Monte Carlo Method* (MCM) is a technique used to approximately solve a mathematical problem through the use of a probabilistic model [Ref. 3: p. 73]. To use the MCM to solve for TGT coefficients, a MATLAB algorithm was developed to "release" a fixed number of "random walkers" from a node on the TGT boundary. The direction in which each of these walkers travel is determined and assigned based on the outcome of a random number generator and the relationship between nodal potentials given by the discretized Laplace equation for the appropriate system. For a z-invariant system, random walkers are assigned an equal 25% probability of going up, down, right or left. This is because the potential of a node in the homogeneous source-free buffer zone between the computational grid and the defined far Dirichlet boundary is 25% of the sum of the potentials of each of its four neighbors (above, below, to the right and left). For rotationally symmetric electrostatic systems, however, the "contributions" of the right and left neighboring nodes are weighted by the $(1 - \frac{1}{2(\frac{r}{\Delta r})})$ and $(1 + \frac{1}{2(\frac{r}{\Delta r})})$ terms of equation (21), repeated below for convenience

$$4\Phi_c = (1 - \frac{1}{2(\frac{r}{\Delta r})})\Phi_l + (1 + \frac{1}{2(\frac{r}{\Delta r})})\Phi_r + \Phi_t + \Phi_b \quad . \tag{31}$$

31

Similarly, the weighting factors for the rotationally symmetric magnetostatic system

come from equation (22), repeated below

$$[4 - \frac{1}{2(\frac{r}{\Delta r} + \frac{1}{2})} + \frac{1}{2(\frac{r}{\Delta r} - \frac{1}{2})}]A_c =$$ (32)

$$(1 - \frac{1}{2(\frac{r}{\Delta r} - \frac{1}{2})})A_l + (1 + \frac{1}{2(\frac{r}{\Delta r} + \frac{1}{2})})A_r + A_t + A_b \ .$$

As can be seen in equation (32), the potential of the center node is also weighted for

magnetostatics.

After every step, each random walker is given a new direction based on a new random

number and the walker's current location. The walkers continue their "journey" until

they land on the computational grid or the distant Dirichlet boundary. If they land on the

Dirichlet boundary, they are eliminated as if to be "nullified" by the zero potential of the

boundary nodes. If they land on the computational grid, a counter which counts the

number of walkers to land on that node is incremented by one and again the walker is

eliminated. Once all of the walkers have been eliminated, the final count of walkers

arriving at each node on the computational grid is divided by the number of walkers

originally released. The end result is a set of coefficients which relate the original walker

release node to each of the nodes on the outside edge of the computational grid. This set

is a row in the TGT matrix. The MATLAB functions which perform this algorithm are

contained in Appendix C. The function *fncylwlk.m* is used for rotationally symmetric electrostatic systems while *fnrctwlk.m* is used for z-invariant systems. The programs *cylcoff.m* and *rctcoff.m*, also in Appendix C, call these functions once for each node on the TGT boundary and assemble the data into the TGT matrix for rotationally symmetric and z-invariant systems respectively. As in the case of the matrix solution method, the TGT coefficients for the nodes on the left side of the TGT boundary in a rotationally symmetric system are derived using symmetry. This is done not just to improve efficiency but out of necessity since a walker on the left edge of the buffer zone is exactly $\frac{\Delta r}{2}$ away from the axis of rotation and by the electrostatic Laplace equation has zero probability of stepping to the left. Accordingly, no walker can ever land on the left side of the computational grid and the TGT relationships can only be determined by using the symmetry of the system. For reasons discussed in the next section, TGT coefficients for rotationally symmetric magnetostatic systems were not calculated using the MCM approach, although this could be easily accomplished with only slight modification to *rctcoff.m* and *fnrctwlk.m*.

D. COMPARISON OF TGT METHODS

Each of the two methods for calculating TGT coefficients discussed previously have advantages and disadvantages. The greatest advantage of the Matrix Solution Method is its accuracy. As discussed in the first section of this chapter, TGT provides the same accuracy as solving the whole open boundary problem without TGT. This is because the matrix solution method uses Poisson's equation to solve for the TGT coefficients just as it would be used to solve the whole open boundary problem. The Monte Carlo Method (MCM) approach however, is only *based* on the Poisson equation and will *always* be subject to an amount of random "noise". The MCM approach generally makes TGT less accurate than solving the whole open boundary problem. For this reason, the TGT coefficients used in EMAG 2.0 were calculated using the matrix solution method.

Insufficient accuracy, however, does not necessarily eliminate the MCM approach because its accuracy is improved by using a very large number of walkers. This increases computation time, but parallel processing can be used to recover this time as a result of the parallel nature of the problem (i.e., the independence of random walk outcomes for different walkers). Further, the MCM approach has the advantage that it allows the user to exchange run time for computer memory. The user can release large sets of walkers a few times to maximize the use of available memory and improve speed of computation, or the user can choose to release only a very few walkers many times thereby increasing computation time but using little memory. In this way, the user can

select a set of walkers of a size which either maximizes or minimizes the use of memory. The number of walkers can effectively be increased by running the program a number of times and averaging the results. In this way, there is no limitation on either the number of walkers or the distance to the Dirichlet boundary except for the CPU time. This can be a distinct advantage over the matrix solution method which is primarily limited by the available memory. The process of layer elimination for the matrix method, requires that the CPU have enough available memory to invert and store a square matrix with dimensions equal to the number of nodes on the first layer to be eliminated. Although sparse matrix tools available in MATLAB help, the inverse of a sparse matrix is generally not sparse. A memory limitation can only be alleviated by moving the Dirichlet boundary closer to the computational grid.

E. TGT OPTIMIZATION

This section summarizes the research into optimization of TGT. The questions to be answered concerning optimization are:

1. Given a computational grid size, is there an optimum distance at which to put the Dirichlet boundary?

2. Using the Monte Carlo Method and the optimum boundary distance from above, how many random walkers are required to produce accurate TGT coefficients?

3. Given the answers to the first two questions, how do speed of computation and memory requirements compare between the two TGT methods?

Optimization of TGT begins with answering the first of these questions. This answer is important because the first step in using TGT is deciding how far away to place the Dirichlet boundary. The result of this decision controls the speed of calculation and memory required as well as the accuracy of calculations within the TGT boundary. In calculating the TGT matrix for use in the original version of EMAG, this decision was made based on the memory limitations of the computer used to calculate the TGT coefficients (a HP-700 series UNIX workstation). As the result, the Dirichlet boundary was established for the "fine" grid to be 803 by 803 nodes or 376 layers away from the computational grid. In order to maintain the same ratio between the boundary and computational grid size, the boundary for the "coarse" grid was made 267 by 267 nodes or 125 layers away from the computational grid [Ref. 1]. The following paragraphs

36

describe the research conducted to determine the optimum boundary location for a given computational grid. This research was conducted using a z-invariant system since the z-invariant computational grid has a distant Dirichlet boundary on all four sides and the layers of nodes in the buffer zone between the boundary and the computational grid are concentric squares. However, it has been verified that the results apply to rotationally symmetric systems as well.

Ideally, if the EMAG computational grid were actually surrounded by free space, the TGT coefficients relating a node on the TGT boundary to the nodes on the outer edge of the computational grid would always sum to unity. In the language of the MCM approach, this is because the walkers would have no outer boundary to land on and would all <u>eventually</u> land somewhere along the edge of the computational grid. As a result of this property, a measure of the quality of the TGT matrix can be defined as

$$Q_{TGT} = \frac{\sum q_i}{n} \quad , \quad q_i = \sum c_n \tag{33}$$

where q_i is the sum of all the coefficients (c_n) relating the i-th node on the TGT boundary to the nodes on the edge of the computational grid and n is the number of nodes on the TGT boundary [Ref. 4]. For a computational grid in infinite free space, Q would be equal to one. Now, if Q was the only factor governing the quality of the TGT coefficients, the original approach of placing the Dirichlet boundary as far away from the

computational grid as possible would be the best solution. However, there is a finite limit at which the size of the Dirichlet boundary becomes so large in comparison to the size of the computational grid, that moving the boundary further away yields diminishing returns. In terms of the MCM approach, this phenomenon can be described as walkers being more likely to land on a very large but distant Dirichlet boundary than a relatively small computational grid simply because of the vast difference in their sizes. As a specific example, an 11 by 11 computational grid would have 40 nodes on its outer edge. A 21 by 21 node Dirichlet boundary layer would be only five layers away but would already have 80 nodes or twice as many as are on the edge of the 11 by 11 computational grid. The size of the Dirichlet boundary grows by eight nodes for every layer it is moved away from the computational grid.

In order to determine the relationship between the size of the computational grid and the optimum size of the Dirichlet boundary, the matrix solution algorithms for calculating TGT coefficients described in section B of this chapter were used. Early attempts at this involved fixing the computational grid size and moving the Dirichlet boundary further away one layer at a time while looking for a point at which the relative change in Q between successive computations was less than 0.1%. During this process, it was noticed that for a fixed Dirichlet boundary size, choosing successively smaller computational grids would eventually result in a maximum value for Q after which it dropped rapidly. This result was contrary to the expectation that more layers of nodes

between the boundary and the computational grid would always result in higher Q's. Through investigation of this phenomenon, it was discovered that for a given Dirichlet boundary size there is a computational grid size which maximizes Q but that the converse of this is not true. This observation led to a "reverse" approach of starting with a known Dirichlet boundary size and searching for the ideal computational grid size. To this end, a Dirichlet boundary size was chosen and the row elimination process involved in calculating the TGT coefficients begun. As each row was eliminated, a new set of TGT coefficients was generated as if the desired TGT boundary had been reached. The Q factor for this set of TGT coefficients was then calculated and stored. This process was continued until the inner-most layer was reached (a 3 by 3 layer of nodes). The stored values for Q were then plotted versus the dimension of the layer at which they were calculated. The resulting curve represented the Q factor trend for a fixed Dirichlet boundary size as the computational grid was decreased in size. The Q curves resulting from using Dirichlet boundary sizes of 51 by 51, 101 by 101 , 151 by 151 and 201 by 201 are shown in Figure 9.
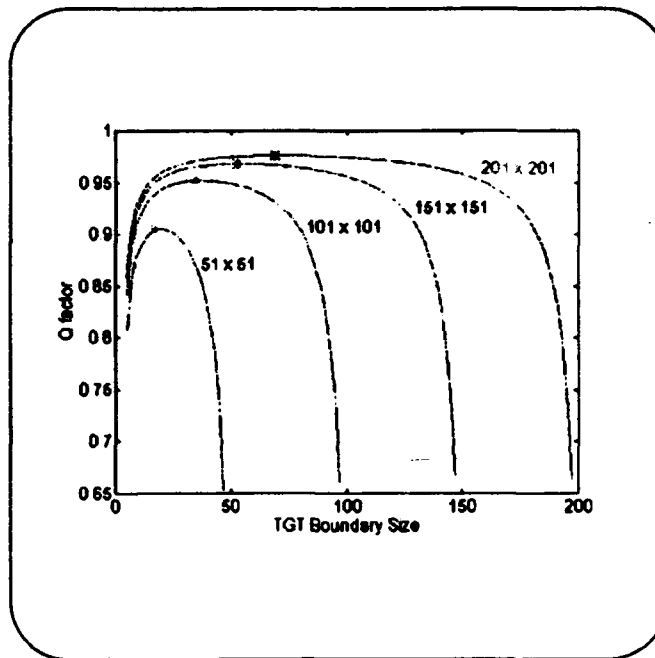
Figure 9. Boundary Curves: Q-Factor Trend

The starred points on each of the curves in Figure 9 represent the TGT boundary size
(and therefore the computational grid size) which corresponds to the maximum value of
Q. These curves illustrate that although moving the boundary further away will always
increase the Q for a fixed computational grid size, this computational grid size will
eventually fall on the left hand side of the boundary curve where Q drops off. Choosing
computational grid sizes which correspond to the maxima on each of these curves results
in a set of TGT coefficients that most effectively match the resolution of the grid. To put
it another way, although moving the Dirichlet boundary further away always improves

40

the TGT coefficients, there is a point at which the resolution of the grid becomes the limiting factor in solution accuracy and improving TGT is no longer beneficial.

To determine the function which relates the optimum computational grid size to the Dirichlet boundary size, boundary curves were generated for 95 Dirichlet boundary sizes ranging from 11 by 11 to 201 by 201. The resulting computational grid sizes were plotted against their corresponding boundary sizes as shown in Figure 10.
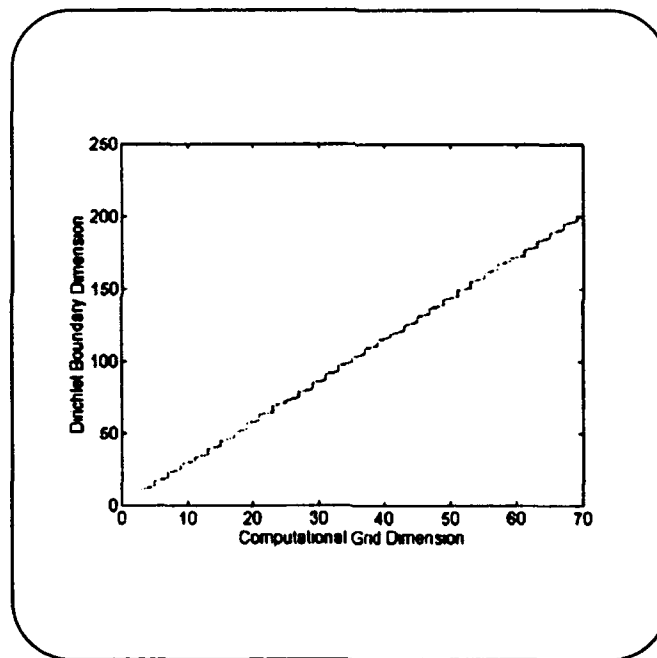
Figure 10. Boundary to Grid Size Relationship

Figure 10 shows that there is a linear relationship between the Dirichlet boundary size and the corresponding optimum computational grid. Figure 11 is a plot of the ratios of the sizes of the data pairs plotted in Figure 10.

41
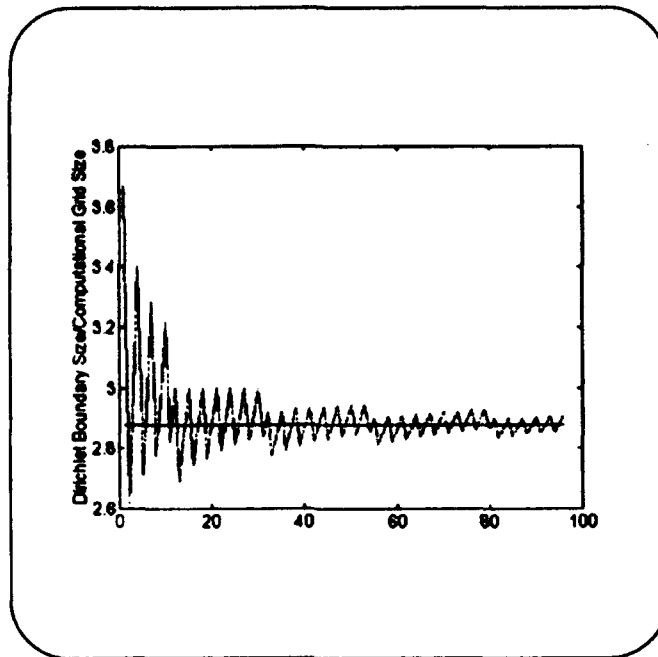
Figure 11. Convergence of Data

Figure 11 shows that the ratio of Dirichlet boundary size to computational grid size

converges to a value of 2.879. As a result, the Dirichlet boundary size to be used with a

given computational grid can be calculated directly by multiplying the desired grid size

by 2.879 and then rounding to the nearest odd integer if the computational grid is of odd

dimension or to the nearest even integer if it is of even dimension.

Once the relationship between the boundary and the computational grid sizes was

determined, the number of MCM random walkers required to accurately solve the TGT

problem could be evaluated. To accomplish this, sets of TGT coefficients were

generated using varying numbers of walkers for a fixed computational grid size and a distant Dirichlet boundary established from the condition given above. These TGT coefficients were then compared to the matrix method TGT coefficients to determine a term-by-term error, ε_k, given by

$$\varepsilon_k = s_k - \tilde{s}_k \qquad (34)$$

where s_k is the k-th, matrix method TGT coefficient and $\tilde{s}_k$ is the k-th coefficient generated by the MCM approach. A root-mean-square (RMS) error was then determined for the MCM coefficients by

$$Error_{RMS} = \sqrt{\frac{\sum(\varepsilon_k)^2}{N}} \qquad (35)$$

where N is the number of coefficients in the set. This result was then used to form an error-to-data ratio by using an RMS measure of the matrix solution TGT coefficient set given by

$$Data_{RMS} = \sqrt{\frac{\sum(s_k)^2}{N}} \qquad (36)$$

43

and the relationship

$$\left(\frac{Error}{Data}\right)_{dB} = 20 \log \left(\frac{Error_{RMS}}{Data_{RMS}}\right) . \tag{37}$$

Using this measure of accuracy, data was collected over a wide range of numbers of walkers. The results of many equal-sized sets of walkers were then used to produce a mean value and standard deviation for the error between trials associated with using a particular number of walkers. The lower curve in Figure 12 depicts the mean error-to-data ratio for sets of MCM walkers using an 11 x 11 computational grid. Curves for one and three standard deviations above the mean are also shown.
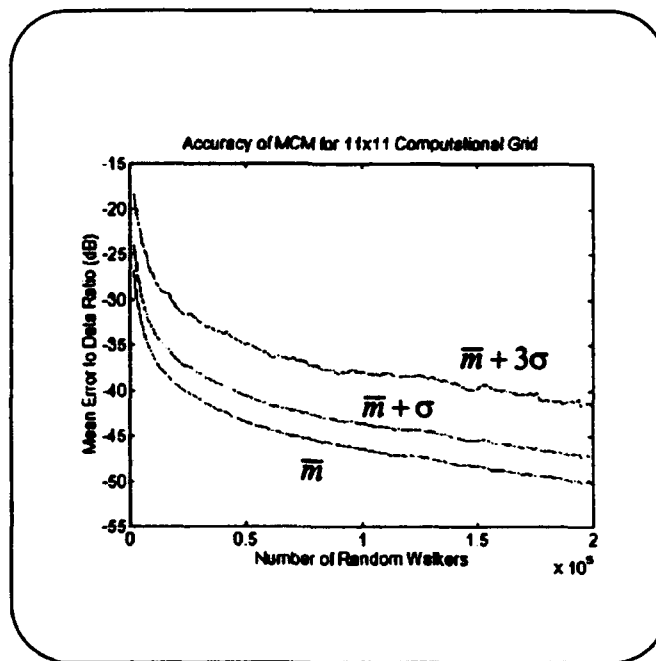
Figure 12. Error vs. Number of Walkers

44

The Central Limit Theorem states that the sum of many statistically independent random variables approaches a Gaussian random variable. The number of walkers that will yield a set of TGT coefficients with a given decibel error 99.7% ($m+3\sigma$) of the time, can therefore be chosen by using the upper standard deviation line in Figure 12 [Ref. 5: pp. 425-430]. To achieve -40dB error with this level of success for an 11 x 11 computational grid would require approximately 1.5 x 10^5 walkers. To achieve this same error level but with 68.3% reliability, the lower standard deviation line ($m+\sigma$) indicates that only about 5 x 10^4 walkers would be required. This type of analysis has been conducted for grid sizes smaller and larger than 11 x 11 and it has been observed that the required number of walkers actually diminishes slightly with increasing grid size (at least up to 51 x 51, the size of EMAG's "fine" grid). This is due to the fact that for larger computational grids a greater number of the TGT coefficients are very small and contribute little to the RMS values for the data and error. An approximation for the $m+\sigma$ curve is given by the equation:

$$(\frac{Error}{Data})_{dB} = 40\,e^{-\frac{x}{10^6}} - 80 \tag{38}$$

where x is the number of walkers released from each node on the TGT boundary. This approximation is only valid for numbers of walkers greater than about 50,000 but gives an indication of how many walkers would be required to achieve accuracies greater than

45

shown in Figure 12. Using this approximation, about 700,000 walkers would be required to achieve a -60 dB error to data ratio with 68.3 % reliability.

With the first two optimization questions answered, the computational speed and memory requirements for the two TGT coefficient generation methods were compared. This comparison was made by calculating a set of TGT coefficients for EMAG's "coarse" and "fine" computational grid sizes. The number of MCM walkers was chosen to provide a -40dB error at a 68.3% level of reliability ($m+\sigma$). Table 1 contains the results of this comparison. The computational times were obtained by calculating TGT coefficients on an IBM compatible 486 class PC. The memory requirements are estimates based only on the sizes of the matrices stored by each coefficient generating program and do not account for any of the MATLAB overhead memory needed for the inversion of the sparse matrices, etc.

TABLE 1: COMPARISON OF TGT METHODS

	17 x 17		51 x 51	
	Memory (KB)	Time (sec)	Memory (KB)	Time (sec)
Matrix Method	271	100	2,700	6,520
MCM with 5×10^4 Walkers	400	29,700	400	217,000

46

As can be seen in Table 1, the matrix solution method is considerably faster than the MCM approach. It should also be noted however, that the memory requirements of the MCM approach do not change with increasing grid size while the memory requirements for the matrix solution method grow rapidly. Furthermore, although computational times for the matrix method are much shorter than the MCM times, they also grow at a much faster rate than the MCM times. Finally, it needs to be noted that the MCM coefficient generating algorithm used here does not take advantage of the symmetry of the system. For z-invariant systems, one could calculate TGT coefficients for one-eighth of the nodes on the TGT boundary and then use symmetry to completely fill the TGT matrix. Such an approach would theoretically cut the MCM times by a factor of eight. Similarly, the rotationally symmetric MCM coefficient generation time can be reduced, but only by a factor of two. Furthermore, only three-quarters as many coefficients need to be generated using MCM for a rotationally symmetric system (the left side coefficients are already obtained by using the symmetry about the axis of rotation) resulting in a computational time of about three-eighths of the value listed in the table.

As shown in this section, there is a choice of method by which the TGT boundary matrix can be calculated. Although the matrix solution method is the most desirable approach when using sequential computers to calculate TGT coefficients for EMAG's current computational grid sizes, the MCM approach can take advantage of parallel processing capabilities. This is because the paths taken by the individual walkers are

statistically independent. The same statistical independence holds even for consecutive steps taken by a single walker. Using a parallel MCM algorithm, a massively parallel computer (1024 processors, for example) could easily be more efficient in calculating TGT coefficients than it would be if it used the matrix solution approach. Further, the MCM approach does provide an effective albeit slow alternative if memory is a limiting factor even on a sequential processing computer. Another important point is that the MCM approach is very intuitive. It has served as a valuable tool in analyzing the generation of the TGT matrix. Finally, using two completely different methods for calculating the same set of coefficients proved to be a tremendous asset in the development and testing of the TGT algorithms contained in the appendices.

IV. EMAG 2.0 EXAMPLES

This chapter presents two examples of EMAG 2.0's solution of rotationally symmetric

problems. These particular problems have been chosen because analytic solutions exist

and can be compared with EMAG's results. The first example is a problem in

electrostatics: the calculation of the capacitance of a cylindrical capacitor. The second

example is a magnetostatics problem which involves calculating the magnetic field on the

axis of a current carrying circular loop.

A. CYLINDRICAL CAPACITOR

The objective of this example is to calculate the capacitance of a cylindrical capacitor as shown in Figure 13. This capacitor has length L = 1 cm, an inner conductor radius a = 0.2 cm, a variable outer conductor radius b, and a polyethylene dielectric separating the two conductors having a relative permittivity $\varepsilon_r = 2.3$.
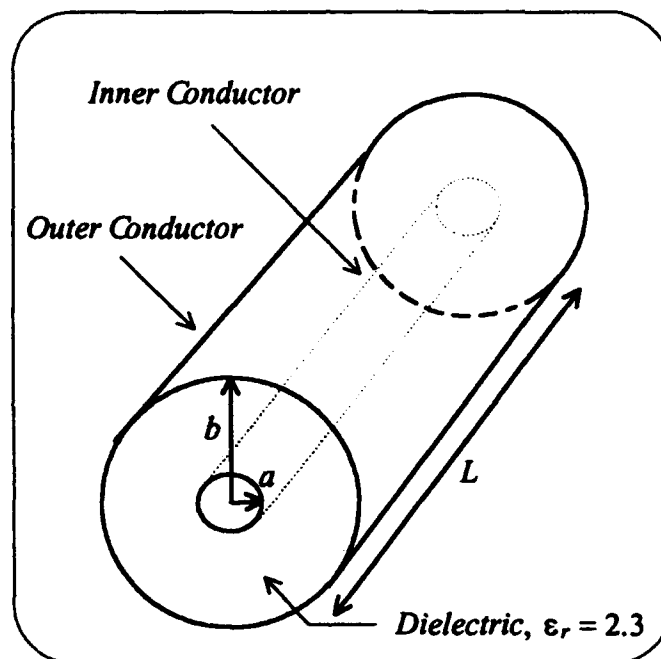
Figure 13. Cylindrical Capacitor

To calculate the capacitance using EMAG 2.0, the capacitor was modeled as shown in Figure 14. EMAG's "enclosed charge" utility was then used to determine the charge on the outer conductor and the result was divided by the known potential difference between the conductors ($V_{outer} - V_{inner}$ = 1 volt) to find the capacitance.

50
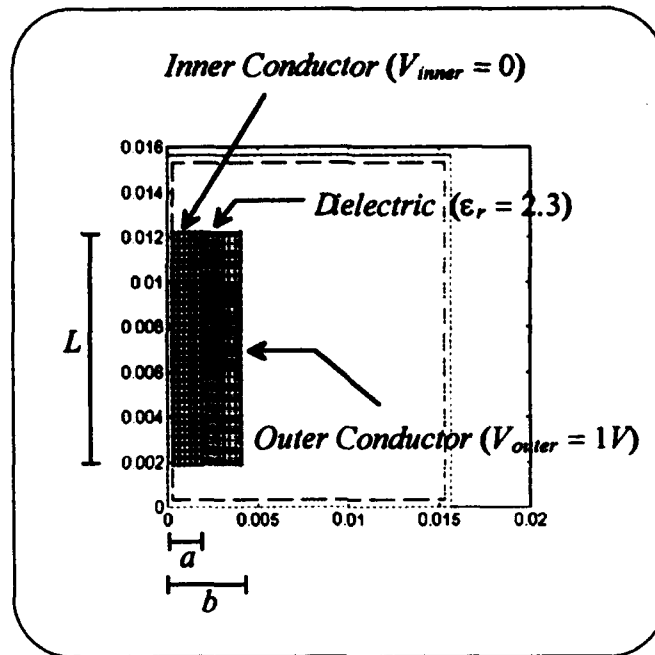
Figure 14. Cylindrical Capacitor

Although Figure 14 depicts a cylindrical capacitor with an outer conductor radius

b = 0.4 cm, Table 2 presents the results for four capacitors with various outer conductor

radii. The relationship used to calculate the theoretical capacitance is

$$C_{Theoretical} = \frac{2\pi\varepsilon_r\varepsilon_oL}{\ln\left(\frac{b}{a}\right)} \; . \tag{39}$$

The derivation of this equation uses Gauss' Law, the definitions of capacitance and

potential, and assumes that the fringing effects near the ends of the capacitor do not

51

contribute to the net capacitance [Ref. 6: p. 125]. Since these end effects are only

negligible when the separation between the conductors is small compared to the length of

the capacitor, the EMAG and theoretical capacitances converge only for long, thin

capacitors. However, end effects cannot be neglected for capacitors in which the length

is not much greater than the conductor separation, and equation (39) is no longer valid.

In this situation, equation (39) can serve only as a lower bound for the actual capacitance.

Table 2 presents the capacitances calculated using EMAG and equation (39), the ratio of

these two values and the separation to length ratio, (b-a)/L, for the four cylindrical

capacitors modeled on EMAG's "fine" computational grid.

TABLE 2: CYLINDRICAL CAPACITOR

b (cm)	C_{EMAG} (pF)	$C_{Theoretical}$ (pF)	$C_{Theoretical}/C_{EMAG}$	(b-a)/L
0.257	5.202	5.103	0.981	0.057
0.300	3.219	3.156	0.980	0.100
0.333	2.842	2.509	0.883	0.133
0.400	2.433	1.990	0.822	0.200

B. MAGNETIC FIELD ALONG AXIS OF A CIRCULAR CURRENT LOOP

This example demonstrates the accuracy of EMAG's coarse and fine grid solvers in
calculating the magnetic field along the axis of a current carrying loop. The problem to
be modeled is shown in Figure 15.
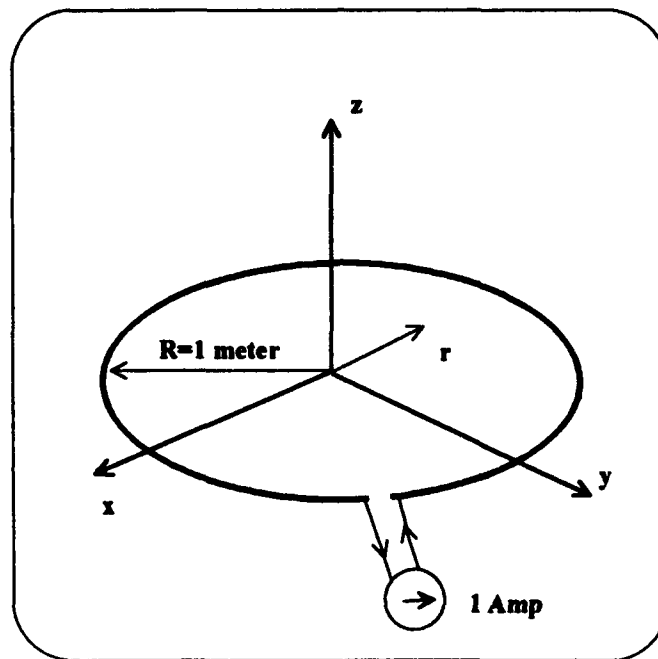
Figure 15. Current Carrying Loop

This problem was modeled in EMAG as a "point" current source on the rotationally

symmetric computational grid. The "point" represented the cross-section of a loop of

current perpendicular to the z-axis. This current was set to 1 Amp. Once this was done,

the magnetic field values corresponding to the nodes on the axis of rotation were

53

extracted from the magnetic field matrix generated by EMAG. The coarse grid model

and the equi-potential contour lines generated by EMAG are shown in Figure 16.
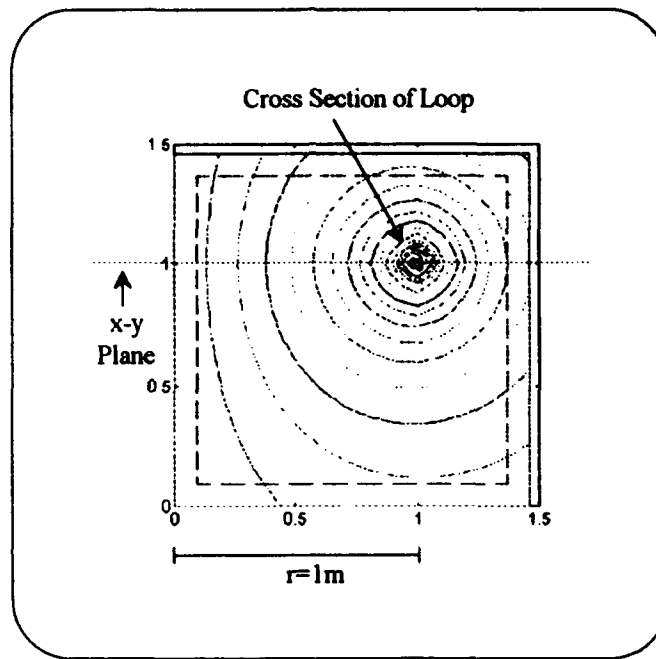
Figure 16. Equal Potential Contours for Current Loop

The fine and coarse grid solutions are plotted in Figure 17 along with the theoretical

field strength given by

$$\vec{H} = \frac{IR^2}{2(r^2 + R^2)^{3/2}} \vec{z}$$

(40)

which was obtained by dividing the theoretical magnetic flux density by the permeability

of free space [Ref. 6 : p. 238].
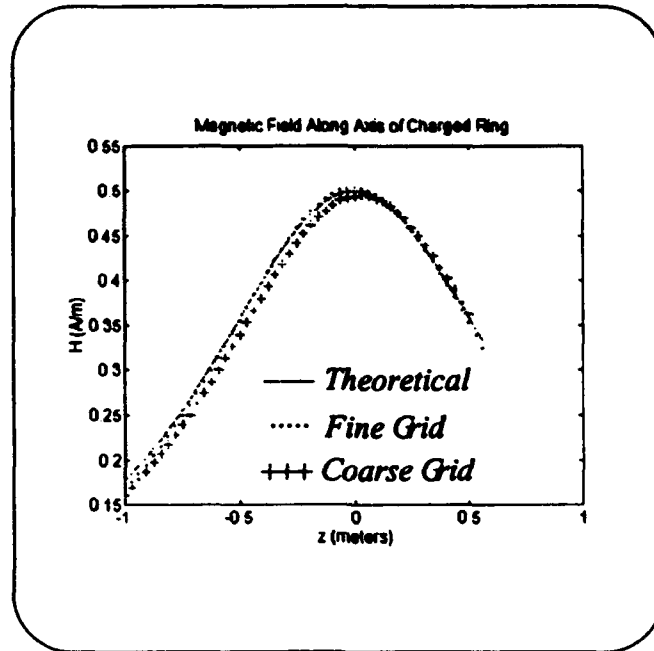
Figure 17. Magnetic Field Along Axis of Loop

As can be seen in Figure 17, EMAG's solutions are nearly identical to the theoretical

solution. Not only does this example demonstrate EMAG's accuracy, but it also shows

how information can be extracted from EMAG to extend the analysis of a given problem.

In this case, the column of data in each of the magnetic field matrices, *hf_fine* and

hf_coarse, which corresponded to the axis of rotation was copied and plotted against the

theoretical solution.

V. CONCLUSIONS

The enhancement of EMAG described in this thesis involved solving two related but separate problems. The first was the generation of finite difference equations for rotationally symmetric electrostatic and magnetostatic systems. The second was the application of these equations in implementing a boundary to accurately simulate infinite free space. In solving the first problem it was found that for rotationally symmetric systems, the discrete Poison equations for electrostatics and magnetostatics are different due to the directional properties of the magnetic vector potential. For z-invariant systems these directional properties can be ignored (because the magnetic vector potential and the current sources are in the z direction). As a result, z-invariant electrostatic and magnetostatic systems can share the same discretized Poisson equation. This is not the case for rotationally symmetric systems. Even if the sources are defined to be exclusively in the θ direction as they are in EMAG 2.0, the directional properties of magnetic vector potential cannot be ignored. The curl operation relating the magnetic field to the magnetic vector potential causes the resulting discrete Poisson equation for magnetostatics to be different than the electrostatic equation developed using the relationship between the electric field and the electric scalar potential which involves a gradient operation.

Once these equations were developed, they were integrated into EMAG and used to calculate boundary relationships based on the concept of TGT. During the course of this work, two different methods were used to calculate the TGT coefficients. Both methods use discretized Poisson equations for a homogeneous, source-free region. Although both approaches create nearly identical sets of TGT coefficients, each have their strengths and weaknesses. The matrix solution approach is more accurate. Using TGT coefficients generated from the matrix approach allows one to solve a problem on a fixed computational grid with the same accuracy as solving a much larger system extending all of the way out to the distant Dirichlet boundary (used to generate the TGT coefficients). The matrix approach was used to determine an optimal relationship between the computational grid size and the size of the distant Dirichlet boundary. This could not have been easily done using the MCM approach because its results are a function of an additional variable: the number of walkers released from each of the TGT nodes. The matrix approach is also the faster of the two approaches, but this would not be the case if a parallel processing computer were used. The major disadvantage of the matrix solution approach is its memory requirement.

The second method for generating TGT coefficients, the MCM approach, uses a probabilistic model in the generation of the TGT coefficients. It can only approach the accuracy of the matrix method when the number of random walkers becomes very large and, on sequential computers, it is also very slow. An extensive parallel processing

57

capability however, could make the MCM approach faster than the matrix approach and therefore, more desirable. The other advantages of the MCM approach are its smaller memory requirements and its intuitive nature.

The end result of above research is a more capable version of EMAG. With its enhancements for rotational symmetry, EMAG can solve a wider set of problems. With the inclusion of TGT generating algorithms, it is possible to modify the program to solve problems on computational grid sizes of the user's choice. The possibilities for future improvements are numerous. They include the enhancement of the graphical user interface, and the addition of the capability to solve time varying problems.

APPENDIX A
EMAG 2.0 LIST OF PROGRAMS

Below is a directory of the files which make up EMAG 2.0. The italicized files are the MATLAB script files which were modified to incorporate rotational symmetry. The following pages contain the complete code listings for these modified files. Code listings for the unmodified files are found in Reference 1.

ccontour.m	helppec.m	pecline3.m
cefield.m	helppost.m	pecpnt.m
chargsrc.m	helpsolv.m	pecreg.m
checkchr.m	helpsrc.m	permat.m
coarsegd.m	*i_calc.m*	plotc.m
connecto.m	in267_17.tgt	source2.m
cy267_17.tgt	in803_51.tgt	table2.m
cy803_51.tgt	*itersoln.m*	plotp.m
cymag_17.tgt	linterp.m	plotq.m
cymag_51.tgt	looktab.m	posterro.m
dielcolo.m	*makesys2.m*	postproc.m
emag.m	*makesysf.m*	printplo.m
epscell.m	matsolve.m	*q_calc.m*
epsreg.m	*matsolvf.m*	redraw2.m
fcontour.m	plotd.m	rprint.m
fefield.m	plotm.m	saveplot.m
fileopt.m	mousetst.m	*solndom.m*
find2rc.m	myquiver.m	solver.m
geosetup.m	nodes.m	table_2.m
hardcop.m	numpec.m	thresh.m
helpemag.m	outline2.m	toggle.m
helpfopt.m	outn51.dat	uavg.m
helpgeo.m	pec.m	voltsrc.m
helpmed.m	peccell.m	xygrido.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                          %
%% cefield.m: Plots either the E-Field or the H-Field depending on the value  %
%%          of the EM_flag, the field vectors can be plotted with the      %
%%          geometry of the problem, and the vectors magnitudes can be      %
%%          "thresholded", or scaled to minimum size as expressed as a      %
%%          percentage of the maximum field.  The field vectors plotted are %
%%          a result of the coarse grid solution.                %
%%          (See thresh.m for more info on threshold)             %
%%                                          %
%%          This program has been modified for rotational symmetry.      %
%%          Magnetic fields are explicitly calculated for rot sym systems.  %
%%          dpw 950515                          %
%%                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

eo_1=pi*4e-7;
geo_on_screen=0;

with_geometry=input('Outline the geometry [y]: ','s');

if strcmp(with_geometry,[]), with_geometry='y'; end

if with_geometry=='y',

   geom_type='c';
   outline2
   hold on

else

   xygrido(xmin,xmax,ymin,ymax,N_coarse,N_coarse)
   hold on

end

ef_coarse=gradient(-v_mati,dx_coarse/3,dy_coarse/3);

if EM_flag=='M'
   uavg
end
```

60

```
if (cyl_flag=='C')&(EM_flag=='M')
   rmat_rowc=0:(length(v_mati)-1);
   rmat_row=rmat_rowc*dx_coarse*(1/3);
   rmatc=rmat_rowc;
   numrows=1;
   while numrows<length(v_mati)
      rmatc=[rmatc;rmat_rowc];
      numrows=numrows+1;
   end
   h_termc=gradient((v_mati.*rmatc),dx_coarse/3,dy_coarse/3);
   h_termc(:,2:length(h_termc))=h_termc(:,2:length(h_termc))./ ...
                    (rmatc(:,2:length(h_termc)));
   h_termc(1:length(h_termc),1)=h_termc(1:length(h_termc),2) ...
                    *(1+(4/3)*dx_coarse/3);

   j=sqrt(-1);
   hf_coarse=(-1)*imag(ef_coarse)+(1j)*real(h_termc);
   hf_coarse=hf_coarse./(eo_1*u_avg_matrix);
   clear rmatc
end

with_thresh=input('Set threshold for the vectors [n]: ','s');

if strcmp(with_thresh,[]), with_thresh='n'; end

if with_thresh=='y',

   hold on

   if EM_flag=='M',

      if cyl_flag=='C'
         [xx,yy]=thresh((-1)*imag(ef_coarse) ./(u_avg_matrix*eo_1), ...
                   real(h_termc) ./(u_avg_matrix*eo_1));
      else
         [yy,xx]=thresh(real(ef_coarse) ./u_avg_matrix *eo_1, ...
                  imag(ef_coarse) ./u_avg_matrix *eo_1);

      end
   else

      [xx,yy]=thresh(real(ef_coarse),-imag(ef_coarse));
```

```
    end

    myquiver(xx,yy,xmax,ymax,'r-');

else

  if EM_flag=='M',

    if cyl_flag=='C'
      myquiver((-1)*imag(ef_coarse) ./(u_avg_matrix*eo_1), ...
             real(h_termc) ./(u_avg_matrix*eo_1),xmax,ymax,'r-');

    else
      myquiver( imag(ef_coarse) ./u_avg_matrix *eo_1, ...
             real(ef_coarse) ./u_avg_matrix *eo_1,xmax,ymax,'r-');
    end

  else

    myquiver(real(ef_coarse),-imag(ef_coarse),xmax,ymax,'r-');

  end

end

hold off

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% end of cefield.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                              %
%% fefield.m: Plots either the E-Field or the H-Field depending on the value  %
%%          of the EM_flag, the field vectors can be plotted with the      %
%%          geometry of the problem, and the vectors magnitudes can be      %
%%          "thresholded" or scaled to minimum size as expressed by a      %
%%          percentage of the maximum field.  The field vectors plotted are %
%%          a result of the fine grid solution. See thresh.m for more info  %
%%                                              %
%%          This file has been modified to explicitly calculate H-field     %
%%          for rotationally symmetric magnetostatic systems.            %
%%          dpw 940515                                  %
%%                                              %
%%                                              %
%%                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


eo_1=pi*4e-7;
geo_on_screen=0;


with_geometry=input('Outline the geometry [y]: ','s');

if strcmp(with_geometry,[]), with_geometry='y'; end

if with_geometry=='y',

  geom_type='f';
  outline2
  hold on

else

  xygrido(xmin,xmax,ymin,ymax,N_fine,N_fine)
  hold on

end

ef_fine=gradient(-v_source_mat,dx,dy); % (-dV/dx)+j(-dV/dy) z-invar
                    % (-dV/dr)+j(-dV/dz) rot sym
```

63

```matlab
if EM_flag=='M'
  uavg
end

if (cyl_flag=='C')&(EM_flag=='M')

  rmat_row=0:(length(v_source_mat)-1);
  rmat_row=(rmat_row)*dx;        % dx=dr
  rmat=rmat_row;
  numrows=1;
  while numrows<length(v_source_mat)
     rmat=[rmat;rmat_row];
     numrows=numrows+1;
  end
  h_term=gradient((v_source_mat.*rmat),dx,dy);
       % h_term=(d(rA)/dr)+j(d(rA)/dz)

  h_term(:,2:length(h_term))=h_term(:,2:length(h_term))./ ...
                (rmat(:,2:length(h_term)));
       % h_term=(1/r)*[(d(rA)/dr)+j(d(rA)/dz)]

  h_term(1:length(h_term),1)=h_term(1:length(h_term),2)*(1+(4/3)*dx);
        % 2.333 (1+4/3) term is from coefficient
        % for values dx from axis
        % This approach avoids division by zero with accurate results.

  j=sqrt(-1);
  hf_fine=(-1)*imag(ef_fine)+(1j)*real(h_term); % mag flux density
         % -1 is due to the fact that
         % MATLAB is taking gradient in -z direction

  hf_fine=hf_fine./(eo_1*u_avg_matrix); % hf_fine is now magnetic field matrix
  clear rmat
end
with_thresh=input('Set threshold for the vectors [n]: ','s');

if strcmp(with_thresh,[]), with_thresh='n'; end

if with_thresh=='y',

  hold on

    if EM_flag=='M',
```

64

```
       if cyl_flag=='C'
         [xx,yy]=thresh((-1)*imag(ef_fine) ./u_avg_matrix, ...
                   real(h_term)  ./u_avg_matrix);

       else
           [yy,xx]=thresh(real(ef_fine) ./u_avg_matrix, ...
                     imag(ef_fine) ./u_avg_matrix);
       end

     else

         [xx,yy]=thresh(real(ef_fine),-imag(ef_fine));

     end

     myquiver(xx,yy,xmax,ymax,'r-');

else

  if EM_flag=='M',

    if cyl_flag=='C'
       myquiver( (-1)*imag(ef_fine) ./u_avg_matrix, ...
             real(h_term)  ./u_avg_matrix,xmax,ymax,'r-');


    else
       myquiver( imag(ef_fine) ./u_avg_matrix, ...
                 real(ef_fine) ./u_avg_matrix,xmax,ymax,'r-');
    end

  else

    myquiver(real(ef_fine),-imag(ef_fine),xmax,ymax,'r-');

  end
end

hold off

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% end of fefield.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                      %
%% i_calc.m: Function to perform a closed line integral of the H-Field along  %
%%       the rectangular contour shown below.                   %
%%                                                      %
%% USAGE: [I]=i_calc(Az,nu,dx,dy)  where: Az is the Magnetic vector potential %
%%                       nu is 1/relative permeability     %
%%       o<---(x1,y1)--------------o  dx and dy are the grid spacing     %
%%       |_|_|_|_|_|_|_|_|_|_|_|_|_|                       %
%%       |_|_|_|Current_|_|_|_|_|_| Az: (MxN) matrix       Units: Wb/m %
%%       |_|_|_|enclosed|_|_|_|_|_| nu: (M-1)x(N-1) matrix  Units: none %
%%       |||||||||||||||| dx and dy: scalar       Units: m   %
%%       o--------------(x2,y2)--->o  I: scalar            Units: A/m %
%%                                                      %
%% When the function is called, the user will use the crosshairs to indicate  %
%% the positions of two opposite corners.  The function will return the      %
%% amount of current enclosed or 'error' if the user has entered the corners  %
%% outside of the region where Az exists or the closed contour contains no    %
%% area. (i.e. the two corners are do not specify a box)            %
%%                                                      %
%%  This file has been modified for rot sym systems. dpw 950515         %
%%                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [i]=i_calc(A,n,dx,dy);
load mouse.emg -mat;

global cyl_flag

disp('')
disp('1. Press the left button for one corner of the box')
disp('2. Press the left button again for the other corner')
disp('3. Press the right button when you are done')
disp('')
disp('If you press the right button immediately after choosing this option')
disp('it will take you back to the previous screen.')
disp('')
%*******************
disp('Press any key to continue...')
pause
%*******************
```

```
%plot;
[xx,yy,button]=ginput(1);

Lower=0;
b1flag=0;
b2flag=0;

while button~=Right_Button | b1flag==0 | b2flag==0,

  if button==Left_Button & Lower==0,
    b1flag=1;
    x(1)=xx;
    y(1)=yy;
    Lower=1;
  elseif button==Left_Button & Lower==1,
    b2flag=1;
    x(2)=xx;
    y(2)=yy;
    Lower=0;
  end

  [xx,yy,button]=ginput(1);

end

x=round(x/dx);
y=round(y/dy);
[r,c]=size(A);

if sum(x<0)==0 & sum(y<0)==0 & abs(x(1)-x(2))>1 & abs(y(1)-y(2))>1 & ...
  sum(x>c)==0 & sum(y>r)==0,

  i=0;
  uo=pi*4e-7;


  if x(1)>x(2), x=[x(2) x(1)]; end
  if y(1)<y(2), y=[y(2) y(1)]; end

  toprow  = r-y(1); bottomrow = r-y(2)-1;
  leftcol = x(1)+1; rightcol  = x(2);

  for col=leftcol+1:rightcol
```

```
      nlr_top   = 0.5*(n(toprow,col-1)   + n(toprow,col));
      nlr_bottom = 0.5*(n(bottomrow,col-1) + n(bottomrow,col));


      i = i + nlr_top   * (A(toprow+1,col) - A(toprow,col)) + ...
          nlr_bottom * (A(bottomrow,col) - A(bottomrow+1,col));

   end

   for row=toprow+1:bottomrow

      ntb_left  = 0.5*(n(row-1,leftcol)  + n(row,leftcol));
      ntb_right = 0.5*(n(row-1,rightcol) + n(row,rightcol));

      if cyl_flag=='C'
         i = i + ntb_left * (1/(leftcol-0.5)) * ...
             ((leftcol)*A(row,leftcol+1)-(leftcol-1)*A(row,leftcol)) ...
             + ntb_right * (1/(rightcol-0.5)) * ...
             ((rightcol-1)*A(row,rightcol)-(rightcol)*A(row,rightcol+1));

      else
         i = i + ntb_left  * (A(row,leftcol+1) - A(row,leftcol)) + ...
             ntb_right * (A(row,rightcol)  - A(row,rightcol+1));
      end
   end

   i=i/uo;
   x(1)=(x(1)+0.5)*dx;
   x(2)=(x(2)-0.5)*dx;
   y(1)=(y(1)-0.5)*dy;
   y(2)=(y(2)+0.5)*dy;
   hold on
   plot([x(1) x(2) x(2) x(1) x(1)],[y(1) y(1) y(2) y(2) y(1)],'c1-')
   hold off

else

   i='error';

end

%%%%%%%%%%%%%%%%%%%%%%%%%% end of i_calc.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                             %
%% itersoln.m:  Used to solve Poisson's equation for E-Statics and M-Statics  %
%%           The equations are formulated using standard Finite Differences%
%%           and the solution algorithm is a essentially Jacobi's Method   %
%%           with or without a approximate starting solution.           %
%%                                             %
%%           This program has been modified from its original version     %
%%           to solve both z-invariant and rotationally symmetric systems %
%%
%%           Modified 940514 dpw
%%                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
cc=[];
if EM_flag=='M',
  prefix='.im';
  hold_er=er_matrix;
  er_matrix=1 ./er_matrix;
  eo_1=pi*4e-7;
else
  prefix='.ie';
  eo_1=1/8.854e-12;
end

float_nodes=[];
num_fnodes=0;
nodes_around_pec=[];
er_around_pec=[];
num_nodes=0;
col_index=0;
obj_index=0;
charge_obj=[];
disp(' ')
disp('Input the %-error you wish to stop at and specify the maximum number')
disp('of iterations.')
disp('When either of these two criteria is satisfied, the program stops')
disp(' ')
TOL=input('Input the error tolerance (in percent) [1]: ');
if strcmp(TOL,[]), TOL=1; end
TOL=TOL/100;
disp(' ')
```

```
MAXITER=input('Input the maximum number of iterations [1000]: ');
if strcmp(MAXITER,[]), MAXITER=1000; end
MAXITER=round(MAXITER/100)*100;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% First, i need to know where all the nodes around all the floating PEC/PMR   %
% objects are ...                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

v_source_mat=9999*ones(ycells+1,xcells+1);

N=N_fine;
ltable=looktab(N);

disp(' ')
disp('Setting up iterative solver variables..')

pntr=find(ltable~=0);

v_source_mat(pntr)=v_source(ltable(pntr));

v_source_mat=reshape(v_source_mat,ycells+1,xcells+1);

for i=1:max(max(v_source_mat))-10003

    fprintf('\nProcessing object #%g - nodes: ',i)

    pts_obj_i=find(v_source_mat==10003+i);

    [row_size,col_size]=size(pts_obj_i);

    if row_size~=0 & col_size~=0,

        obj_index=obj_index+1;

        col_index=col_index+1;

        float_nodes(1:row_size,col_index)=pts_obj_i;
        num_fnodes(col_index)=row_size;

        row_index=1;
```

```
num_nodes(obj_index)=0;

for j=1:row_size

    fprintf('%g ',j)

    c_row=rem(pts_obj_i(j),ycells+1);
    c_col=ceil(pts_obj_i(j)/(xcells+1));
    t_row=c_row-1; t_col=c_col;
    b_row=c_row+1; b_col=c_col;
    l_row=c_row;  l_col=c_col-1;
    r_row=c_row;  r_col=c_col+1;


    % look up to see if there is no PEC/PMR

    if v_source_mat(t_row,t_col)==9999,

       nodes_around_pec(row_index,col_index)=(t_col-1)*(ycells+1)+t_row;
       er_around_pec(row_index,col_index)=(er_matrix(c_row-1,c_col-1)+ ...
                                   er_matrix(c_row-1,c_col))/2;
       num_nodes(obj_index)=num_nodes(obj_index)+1;
       row_index=row_index+1;
    end

    % look left to see if there is no PEC/PMR

    if v_source_mat(l_row,l_col)==9999,

       nodes_around_pec(row_index,col_index)=(l_col-1)*(ycells+1)+l_row;
       er_around_pec(row_index,col_index)=(er_matrix(c_row-1,c_col-1)+ ...
                                   er_matrix(c_row,c_col-1))/2;
       num_nodes(obj_index)=num_nodes(obj_index)+1;
       row_index=row_index+1;
    end

    % look right to see if there is no PEC/PMR
    if v_source_mat(r_row,r_col)==9999,

       nodes_around_pec(row_index,col_index)=(r_col-1)*(ycells+1)+r_row;
       er_around_pec(row_index,col_index)=(er_matrix(c_row-1,c_col)+ ...
                                   er_matrix(c_row,c_col))/2;
       num_nodes(obj_index)=num_nodes(obj_index)+1;
```

71

```matlab
                row_index=row_index+1;
            end

            % look down to see if there is no PEC/PMR

            if v_source_mat(b_row,b_col)==9999,

                nodes_around_pec(row_index,col_index)=(b_col-1)*(ycells+1)+b_row;
                er_around_pec(row_index,col_index)=(er_matrix(c_row,c_col-1)+ ...
                                            er_matrix(c_row,c_col))/2;
                num_nodes(obj_index)=num_nodes(obj_index)+1;
                row_index=row_index+1;
            end

        end

        fprintf('\n')

    end

end

% calculate the total charge/current for each PEC/PMR object

for i=1:obj_index
    charge_obj(i)=sum(charge_mat(float_nodes(1:num_fnodes(i),i)));
    charge_mat(float_nodes(1:num_fnodes(i),i))=zeros(1,num_fnodes(i));
    disp(' ')
    disp(['Charge for object #',int2str(i),': ',num2str(charge_obj(i)),' C/m'])
    disp(' ')
end

charge_obj=charge_obj*eo_1;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Set up pointers to the middle and outer layers              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if cyl_flag=='C'
 if EM_flag=='M'
    load cymag_51.tgt -mat  % coefficients for mag rot sym
 else
```

```
    load cy803_51.tgt -mat  % coefficients for elect rot sym
  end
else
  load in803_51.tgt -mat  % coefficients for z-invariance
end
N=N_fine+1;
xmat=table_2(N);

%clear middle_pts outer_pts

disp('')
disp('Initializing pointers to the outer and middle layers')

if exist('outN51.dat')~=2,
  for i=1:2*(xcells+ycells)
    outer_pts(i)=find(xmat==i);
  end
  save outN51.dat outer_pts
else
  load outN51.dat -mat
end
if exist('midN51.dat')~=2,
  for i=2*(xcells+ycells)+1:2*(xcells+ycells)+2*(xcells-2+ycells-2)
    middle_pts(i-120)=find(xmat==i);
  end
  save mid51.dat middle_pts
else
  load midN51.dat -mat
end

disp('')
disp('Set source pointers and media pointers')

source_pts=find(v_source_mat@9999);

v_known=v_source_mat(source_pts);

diel_pts=find(v_source_mat>=9999);
[drow,dcol]=size(diel_pts);
v_source_mat(diel_pts)=zeros(1,drow);

try_again='y';
load_success=0;
```

```matlab
suffix='x';
ig_loaded='n';

% to load or not load an initial solution

init_guess=input('Do you wish to use an initial guess? [y]: ','s');
if strcmp(init_guess,[]), init_guess='y'; end

if init_guess=='y',

  if exist('v_mat')==1,
    disp(' ')
    ig_loaded=input('Use current coarse grid solution? [y]: ','s');
    if strcmp(ig_loaded,[]), ig_loaded='y'; end
  end

  if ig_loaded=='y',

    v_source_mat=linterp(v_mat);

  else

    while try_again=='y',
        disp(' ')
        disp(' Enter the file which contains the initial guess: ')
        fname=input('with no file extension: ','s');
        while suffix~='c' & suffix~='f',
          disp(' ')
        disp('Is the data from a coarse or fine grid solution? ')
          suffix=input('Enter c or f: ','s');
        end

        if exist([fname,prefix,suffix])==2,
          eval(['load ',fname,prefix,suffix,' -mat']);
          try_again='n';
          load_success=1;
        else
          disp(' ')
        disp([bell,fname,prefix,suffix, ...
                          ' does not exist in the current directory!'])
          suffix='x';
          try_again=input('Do you wish to try again? [y]: ','s');
          if strcmp(try_again,[]), try_again='y'; end
```

74

```
        end

    end

    if load_success==1 & suffix=='c',
        v_source_mat=linterp(v_mat);
    elseif load_success==0,
        disp(' ')
      disp([bell,'An initial guess of all zeros will be used', ...
                                          ' (except known potentials)!']);
        v_source_mat=zeros(N_fine+1,N_fine+1);
        v_source_mat(source_pts)=v_known;

    end

  end

else
  disp(' ')
  disp([bell,'An initial guess of zeros will be used', ...
                                        ' (except known potentials)!'])
  v_source_mat=zeros(N_fine+1,N_fine+1);
  v_source_mat(source_pts)=v_known;

end



mesh (flipud(v_source_mat))

disp('')
disp('Averaging media properties')

r=(2:ycells)';
c=(2:xcells)';

rt=r-1;
rb=r+1;
cl=c-1;
cr=c+1;

if cyl_flag=='C'    % rotational symmetry
  cols=1:length(c);
```

```matlab
for k=1:length(r)
 cc=[cc;cols];
end
if EM_flag=='M'
  erl(r,c)=(er_matrix(r-1,c-1)+er_matrix(r,c-1)).*(1-(2*cc-1).^(-1));
  err(r,c)=(er_matrix(r-1,c)+er_matrix(r,c)).*(1+(2*cc+1).^(-1));
  ert(r,c)=er_matrix(r-1,c-1)+er_matrix(r-1,c);
  erb(r,c)=er_matrix(r,c-1)+er_matrix(r,c);
  erc(r,c)=2*(er_matrix(r-1,c-1) ...
        +er_matrix(r,c-1)).*(1+(1/2)*(2*cc-1).^(-1)) ...
        +2*(er_matrix(r-1,c) ...
        +er_matrix(r,c)).*(1-(1/2)*(2*cc+1).^(-1));
        %% 2 term is from 1/2 avg of reluctivities
else
  erl(r,c)=(er_matrix(r-1,c-1)+er_matrix(r,c-1)).*(1-(2*cc-1).^(-1));
  err(r,c)=(er_matrix(r-1,c)+er_matrix(r,c)).*(1+(2*cc-1).^(-1));
  ert(r,c)=er_matrix(r-1,c-1).*(1-(4*cc-2).^(-1)) ...
        +er_matrix(r-1,c).*(1+(4*cc-2).^(-1));
  erb(r,c)=er_matrix(r,c-1).*(1-(4*cc-2).^(-1)) ...
        +er_matrix(r,c).*(1+(4*cc-2).^(-1));
  erc(r,c)=(er_matrix(r-1,c-1)+er_matrix(r,c-1)).*(2-3*(4*cc-2).^(-1)) ...
        +(er_matrix(r-1,c)+er_matrix(r,c)).*(2+3*(4*cc-2).^(-1));
end

else            % z-invariance
 erl(r,c)=er_matrix(r-1,c-1)+er_matrix(r,c-1);
 err(r,c)=er_matrix(r-1,c)+er_matrix(r,c);
 ert(r,c)=er_matrix(r-1,c-1)+er_matrix(r-1,c);
 erb(r,c)=er_matrix(r,c-1)+er_matrix(r,c);
 erc(r,c)=2*(erl(r,c)+err(r,c)); % 2 term is from 1/2 of average of epsilons
                                 % from left, right, bottom and top terms!!!
end
derr=err(r,c) ./ erc(r,c);
derl=erl(r,c) ./ erc(r,c);
dert=ert(r,c) ./ erc(r,c);
derb=erb(r,c) ./ erc(r,c);

dcharge_mat=2*eo_1*charge_mat(r,c) ./ erc(r,c);

outer_pts=outer_pts';
middle_pts=middle_pts';
```

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The main loop of the iterative algorithm                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
disp(' ')
disp('Starting the iterative solver now ... HEAVY NUMBER CRUNCHING')

N=100;
error=0;
direction=+1;  % error is going up instead of down.
done='n';
iterations=0;
errordat=[];
sinfdat=[];
while done=='n',

    while iterations < MAXITER & (error > TOL | (error <= TOL & direction==1) ),

        % for 96 iterations do not calculate the norm s(n), becuase it is an
        % expensive operation, do this for the last four iterations of every
        % 100 iterations

            for q=1:96

            % apply Poission's equation to every point

            v_source_mat(r,c)=dcharge_mat + v_source_mat(r,cr).*derr + ...
                        v_source_mat(r,cl).*derl+v_source_mat(rb,c).*derb+ ...
                        v_source_mat(rt,c).*dert;

            % reset the source points back to their known values

            v_source_mat(source_pts)=v_known;

            % for each floating PEC/PMR object enforce the appropriate boundary
            % conditions, flux of D equals zero and equipotential (for E-statics)
            % or H normal equals zero, equipotential, and curl of H equals zero
            % (for M-statics)

            for j=1:obj_index

                n_num=num_nodes(j);
```

```
          f_num=num_fnodes(j);
          e_a_p=er_around_pec(1:n_num,j);

      v_source_mat(float_nodes(1:f_num,j))=ones(1,f_num)*(charge_obj(j) + ...
      v_source_mat(nodes_around_pec(1:n_num,j))'* e_a_p)/ sum(e_a_p);

      end

% fix the boundary nodes with the TGT matrix to simulate the "open"
      % boundary

      v_source_mat(outer_pts)=inside*v_source_mat(middle_pts);

  end

% for the last four iterations of every 100, calcualate s(n)

for q=1:4

      v_source_mat(r,c)=dcharge_mat + v_source_mat(r,cr).*derr + ...
                  v_source_mat(r,cl).*derl+v_source_mat(rb,c).*derb+ ...
                  v_source_mat(rt,c).*dert;

      v_source_mat(source_pts)=v_known;

      for j=1:obj_index

        n_num=num_nodes(j);
        f_num=num_fnodes(j);
        e_a_p=er_around_pec(1:n_num,j);

      v_source_mat(float_nodes(1:f_num,j))=ones(1,f_num)*(charge_obj(j) + ...
      v_source_mat(nodes_around_pec(1:n_num,j))'* e_a_p)/ sum(e_a_p);

      end

      v_source_mat(outer_pts)=inside*v_source_mat(middle_pts);

      s(q)=sum(sum(v_source_mat .^2));

  end
```

```matlab
% using the last three norms (s(n)), calculate the lambda coefficient and
% the estimated norm as iterations->infinity to give a more accurate
% determination of the actual error

lambda1=(s(3)-s(2))/(s(2)-s(1));
s_infinity1=(s(3)-lambda1*s(2))/(1-lambda1);

lambda2=(s(4)-s(3))/(s(3)-s(2));
s_infinity2=(s(4)-lambda2*s(3))/(1-lambda2);

s_infinity=(s_infinity1+s_infinity2)/2;

olderror=error;

error=abs((s(4)-s_infinity)/s(4));

errordat=[errordat error];
sinfdat=[sinfdat s_infinity];

direction=sign(error-olderror);

iterations=iterations+100;
disp(' ')
disp(['Estimated Error: ',num2str(error*100), ...
      ' Iterations: ',int2str(iterations), ...
      ' S_inf: ',num2str(s_infinity)])

end

mesh(flipud(v_source_mat))

done='y';

if error@=TOL,
  disp(' ')
  disp([bell,'The specified error (',num2str(TOL*100), ...
                                    '%) has been reached.'])
  disp(' ')
  disp(['The current number of iterations: ',int2str(iterations)])

end

if iterations>=MAXITER,
```

79

```
    disp(' ')
    disp([bell,'The maximum number of iterations has been reached, ', ...
                        int2str(MAXITER),' iterations.'])
    disp(' ')
    disp(['The current estimated error is: ',num2str(error*100),'%.']);


end

disp(' ')
newerror=input('Do you wish to modify this? [n] ','s');
if strcmp(newerror,[]), newerror='n'; end
if newerror=='y',

  done='n';

  while (error<=TOL | iterations>=MAXITER) & done=='n',

      done='n';
    disp(' ')
      TOL=input('New error tolerance (%): ')/100;
      MAXITER=input('New maximum # of iterations (UNITS OF 100!): ')*100;

      if error<=TOL | iterations>=MAXITER,
      disp(' ')
        disp([bell,'One or both of those conditions is already ', ...
                    'satisfied! '])
      disp(' ')
        done=input('Do you wish to try again? [y]','s');

        if done=='n',
          done='y';
        else
          done='n';
        end

      else

        disp(' ')
        disp('Continuing iterative solver ...')

      end
```

```
    end

  end

end

if EM_flag=='M',
  er_matrix=hold_er;
  eo_1=1/8.854e-12;
end

clear dcharge_mat derr dert derb derl f_num n_num e_a_p

%%%%%%%%%%%%%%%%%%%%%%%%%%%%% end of itersoln.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                                %
%% makesys2.m: This program will generate the system matrix for the coarse   %
%%        grid using the geometry media and the TGT coefficients.  This %
%%        system matrix sys_mat will then be modified by matsolve.m      %
%%        using the appropriate source and object boundary conditions.  %
%%                                                %
%%        This program has been modified to generate the coarse grid     %
%%        system matrix for both z-invariant and rotationally symmetric %
%%        systems. dpw 940514                       %
%%                                                %
%%        Note that different code is required for rotationally       %
%%        symmetric electrostatic and magnetostatic problems.        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if EM_flag=='M',              % for M-statics the er_matrix
   hold_er=er_matrix_coarse;           % hold the value of relative
   er_matrix_coarse=1 ./ er_matrix_coarse;   % permeability.
   eo_1=pi*4E-7;
end

if cyl_flag=='C'        % rotationally symmetric system

   if EM_flag=='M'
      load cymag_17.tgt -mat   % tgt matrix for magnetostatics (rot sym)
   else
      load cy267_17.tgt -mat   % tgt mat for electrostatics (rot sym)
   end

else              % z-invariant system
   load in267_17.tgt -mat
end

N=N_coarse;       % # of cells for coarse grid
ltable=looktab(N);   % call lookup table routine

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Now, make the matrix!!!!                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```matlab
sys_mat=zeros((N+1)*(N+1)-4,(N+1)*(N+1)-4);  % "-4" is to exclude the four
rhs=zeros((N+1)*(N+1)-4,1);                    % corners in the system matrix

for eq_num=4*N-3:(N+1)*(N+1)-4   % equation # starts at 4N-3 becuase the first
                   % 4N-4 equations come from the TGT boundary
                   % condition matrix
  pntr=find(ltable==eq_num);  %point to the grid point for equation #: eq_num

  c_row=rem(pntr,N+1);      % current row and column of geometry converted
  c_col=ceil(pntr/(N+1));   % from the "find" commmand's indexing scheme

  t_row=c_row-1;,  t_col=c_col;  % top row and column
  b_row=c_row+1;,  b_col=c_col;  % bottom row and column
  l_row=c_row;,    l_col=c_col-1; % left row and column
  r_row=c_row;,    r_col=c_col+1; % right row and column

  t_num=(t_col-1)*(N+1)+t_row;   % convert row and column info of the above
  b_num=(b_col-1)*(N+1)+b_row;   % variables back into the "find"-indexing
  l_num=(l_col-1)*(N+1)+l_row;   % scheme
  r_num=(r_col-1)*(N+1)+r_row;

% pre-calculate the averaging of the media

%% Rotationally Symmetric Systems
if cyl_flag=='C'       % rotationally symmetric systems
 if EM_flag=='M'
     er_top =  er_matrix_coarse(c_row-1,c_col-1) ...
           +er_matrix_coarse(c_row-1,c_col);
     er_bottom =er_matrix_coarse(c_row,c_col-1) ...
           +er_matrix_coarse(c_row,c_col);
     er_left  =(er_matrix_coarse(c_row-1,c_col-1) ...
           +er_matrix_coarse(c_row,c_col-1)) ...
           *(1-1/(2*(c_col-1)-1));
     er_right  =(er_matrix_coarse(c_row-1,c_col) ...
           +er_matrix_coarse(c_row,c_col)) ...
           *(1+1/(2*(c_col-1)+1));
     er_center =(er_matrix_coarse(c_row-1,c_col) ...
           +er_matrix_coarse(c_row,c_col)) ...
           *(1-(1/2)*(1/(2*(c_col-1)+1))) ...
           +(er_matrix_coarse(c_row-1,c_col-1) ...
           +er_matrix_coarse(c_row,c_col-1)) ...
           *(1+(1/2)*(1/(2*(c_col-1)-1)));
```

```
    else
        a_col=c_col-1;        % actual column away from centerline
        er_top  =er_matrix_coarse(c_row-1,c_col-1)*(1-1/(4*a_col-2)) ...
                +er_matrix_coarse(c_row-1,c_col )*(1+1/(4*a_col-2));
        er_bottom=er_matrix_coarse(c_row,c_col-1)*(1-1/(4*a_col-2)) ...
                +er_matrix_coarse(c_row,c_col )*(1+1/(4*a_col-2));
        er_left =(er_matrix_coarse(c_row-1,c_col-1) ...
                +er_matrix_coarse(c_row,c_col-1))*(1-1/(2*a_col-1));
        er_right =(er_matrix_coarse(c_row-1,c_col) ...
                +er_matrix_coarse(c_row,c_col))*(1+1/(2*a_col-1));
        er_center=(er_matrix_coarse(c_row-1,c_col-1) ...
                +er_matrix_coarse(c_row,c_col-1))*(1-3/(8*a_col-4)) ...
                +(er_matrix_coarse(c_row-1,c_col) ...
                +er_matrix_coarse(c_row,c_col))*(1+3/(8*a_col-4));
    end

else   %% z-invariant systems
    er_top  =er_matrix_coarse(c_row-1,c_col-1)+er_matrix_coarse(c_row-1,c_col);
    er_bottom=er_matrix_coarse(c_row,c_col-1) +er_matrix_coarse(c_row,c_col);
    er_left =er_matrix_coarse(c_row-1,c_col-1)+er_matrix_coarse(c_row,c_col-1);
    er_right =er_matrix_coarse(c_row-1,c_col) +er_matrix_coarse(c_row,c_col);
    er_center=er_top+er_bottom;

end
% fill the system matrix!

sys_mat(eq_num,eq_num)      =-2*er_center;
sys_mat(eq_num,ltable(t_num))=er_top;
sys_mat(eq_num,ltable(b_num))=er_bottom;
sys_mat(eq_num,ltable(l_num))=er_left;
sys_mat(eq_num,ltable(r_num))=er_right;

% generate the Right hand side forcing function

rhs(eq_num)=-2*charge_mat_coarse(c_row,c_col)*eo_1;

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Modify the inside matrix of tgt coefficients to exclude the 4 corners      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
insidep=[inside(2:N,:);
      inside(N+2:2*N,:);
      inside(2*N+2:3*N,:);
      inside(3*N+2:4*N,:)];

insidepp=[inside(1,:);
      inside(N+1,:);
      inside(2*N+1,:);
      inside(3*N+1,:)];

[r_inp,c_inp]=size(insidep);

sys_mat(1:r_inp,r_inp+1:r_inp+c_inp)=insidep;
sys_mat(1:r_inp,1:r_inp)=-eye(r_inp);


if EM_flag=='M',
  er_matrix_coarse=hold_er;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% end of makesys2.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%  This will be used to augment the iterative solver by using
%%  sparse matrices to solve fine grid system. Updated 940514 dpw
%%
%% makesysf.m: This program will generate the system matrix for the fine
%%          grid using the geometry media and the TGT coefficients.  This
%%          system matrix sys_mat will then be modified by matsolvf.m
%%          using the appropriate source and object boundary conditions.
%%          This program handles either rotationally symmetric or
%%          z-invariant systems.
%%
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if EM_flag=='M',                    % for M-statics the er_matrix
  hold_er=er_matrix;                % hold the value of relative
  er_matrix=1 ./ er_matrix;         % permeability.
  eo_1=pi*4E-7;
end

if cyl_flag=='C'            % rotationally symmetric system

  if EM_flag=='M'
    load cymag_51.tgt -mat
  else
    load cy803_51.tgt -mat
  end

else                    % z-invariant system
  load in803_51.tgt -mat
end

N=N_fine;        % # of cells for coarse grid
ltable=looktab(N);   % call lookup table routine

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Now, make the matrix!!!!                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
sys_mat=sparse((N+1)*(N+1)-4,(N+1)*(N+1)-4);  % "-4" is to exclude the four
rhs=sparse((N+1)*(N+1)-4,1);                   % corners in the system matrix

for eq_num=4*N-3:(N+1)*(N+1)-4   % equation # starts at 4N-3 becuase the first
                     % 4N-4 equations come from the TGT boundary
                     % condition matrix
  pntr=find(ltable==eq_num);  %point to the grid point for equation #: eq_num


  c_row=rem(pntr,N+1);        % current row and column of geometry converted
  c_col=ceil(pntr/(N+1));     % from the "find" commmmand's indexing scheme


  t_row=c_row-1;,  t_col=c_col;  % top row and column
  b_row=c_row+1;,  b_col=c_col;  % bottom row and column
  l_row=c_row;,    l_col=c_col-1; % left row and column
  r_row=c_row;,    r_col=c_col+1; % right row and column


  t_num=(t_col-1)*(N+1)+t_row;   % covert row and column info of the above
  b_num=(b_col-1)*(N+1)+b_row;   % variables back into the "find"-indexing
  l_num=(l_col-1)*(N+1)+l_row;   % scheme
  r_num=(r_col-1)*(N+1)+r_row;

% pre-calculate the averaging of the media

if cyl_flag=='C'      % rotationally symmetric system
 if EM_flag=='M'
    er_top =  er_matrix(c_row-1,c_col-1) ...
            +er_matrix(c_row-1,c_col);
    er_bottom =er_matrix(c_row,c_col-1) ...
            +er_matrix(c_row,c_col);
    er_left  =(er_matrix(c_row-1,c_col-1) ...
            +er_matrix(c_row,c_col-1)) ...
            *(1-1/(2*(c_col-1)-1));
    er_right  =(er_matrix(c_row-1,c_col) ...
            +er_matrix(c_row,c_col)) ...
            *(1+1/(2*(c_col-1)+1));
    er_center =(er_matrix(c_row-1,c_col) ...
            +er_matrix(c_row,c_col)) ...
            *(1-(1/2)*(1/(2*(c_col-1)+1))) ...
            +(er_matrix(c_row-1,c_col-1) ...
            +er_matrix(c_row,c_col-1)) ...
            *(1+(1/2)*(1/(2*(c_col-1)-1)));
```

```matlab
    else
        a_col=c_col-1;        % actual column away from centerline
        er_top   =er_matrix(c_row-1,c_col-1)*(1-1/(4*a_col-2)) ...
                +er_matrix(c_row-1,c_col )*(1+1/(4*a_col-2));
        er_bottom=er_matrix(c_row,c_col-1)*(1-1/(4*a_col-2)) ...
                +er_matrix(c_row,c_col )*(1+1/(4*a_col-2));
        er_left  =(er_matrix(c_row-1,c_col-1) ...
                +er_matrix(c_row,c_col-1))*(1-1/(2*a_col-1));
        er_right =(er_matrix(c_row-1,c_col) ...
                +er_matrix(c_row,c_col))*(1+1/(2*a_col-1));
        er_center=(er_matrix(c_row-1,c_col-1) ...
                +er_matrix(c_row,c_col-1))*(1-3/(8*a_col-4)) ...
                +(er_matrix(c_row-1,c_col) ...
                +er_matrix(c_row,c_col))*(1+3/(8*a_col-4));
    end
  else %% z-invariant system
    er_top   =er_matrix(c_row-1,c_col-1)+er_matrix(c_row-1,c_col);
    er_bottom=er_matrix(c_row,c_col-1) +er_matrix(c_row,c_col);
    er_left  =er_matrix(c_row-1,c_col-1)+er_matrix(c_row,c_col-1);
    er_right =er_matrix(c_row-1,c_col) +er_matrix(c_row,c_col);
    er_center=er_top+er_bottom;

  end
  % fill the system matrix!

  sys_mat(eq_num,eq_num)      =-2*er_center;
  sys_mat(eq_num,ltable(t_num))=er_top;
  sys_mat(eq_num,ltable(b_num))=er_bottom;
  sys_mat(eq_num,ltable(l_num))=er_left;
  sys_mat(eq_num,ltable(r_num))=er_right;

  % generate the Right hand side forcing function

  rhs(eq_num)=-2*charge_mat(c_row,c_col)*eo_1;

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Modify the inside matrix of tgt coefficients to exclude the 4 corners      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
insidep=[inside(2:N,:);
     inside(N+2:2*N,:);
     inside(2*N+2:3*N,:);
     inside(3*N+2:4*N,:)];

insidepp=[inside(1,:);
     inside(N+1,:);
     inside(2*N+1,:);
     inside(3*N+1,:)];

[r_inp,c_inp]=size(insidep);

sys_mat(1:r_inp,r_inp+1:r_inp+c_inp)=insidep;
sys_mat(1:r_inp,1:r_inp)=-eye(r_inp);


if EM_flag=='M',
  er_matrix=hold_er;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% end of makesysf.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  This will be used to augment the iterative solver by
%%  using sparse matrices to slve the fine grid system.
%%  Updated 940107 dpw.
%%
%%
%% matsolvf.m: This program will use the sys_mat from makesysf.m and the
%%             source information from voltsrc.m and chargsrc.m and and solve
%%             the system of equations for the fine grid.
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

k=(xcells+1)*(ycells+1)-4;
v_pntr=1:k;
ep_rem_col_flag=[];
kp_rem_col_flag=[];
keep_col_flag=[];
keep_row_flag=[];

% find all the floating PEC/PMR and eliminate the need to solve for them, only
% need to solve for one of the potentials for each object

for i=1:num_pec_obj

  equal_pots=find(v_source==10000+3+i);

  [dum,num_eq_pots]=size(equal_pots);

  while num_eq_pots>1,

    % adding the rows and columns of the equal pots and set the first
    % row and column number (equal_pot(1)) to the sum, also add the charges
    % on the floating objects



    sys_mat(:,equal_pots(1))=sum(sys_mat(:,equal_pots)')';
    sys_mat(equal_pots(1),:)=sum(sys_mat(equal_pots,:));
    rhs(equal_pots(1))=sum(rhs(equal_pots));

    % set the equi-potential remove column flag to the rows and column
```

90

% corresponding to the floating potentials we don't need to solve for
% anymore.

    ep_rem_col_flag=[ep_rem_col_flag equal_pots(2:num_eq_pots)];

    v_pntr(equal_pots(2:num_eq_pots))=equal_pots(1)*ones(1,num_eq_pots-1);

    num_eq_pots=0;

  end

end

% find all the known potentials

vs=zeros(1,k);
known_pots=find(v_source@9999);
[dum,num_kn_pots]=size(known_pots);
vs(known_pots)=v_source(known_pots);

kp_rem_col_flag=known_pots;

for i=1:k

  % if either of the flags below contain "i" then keep that row and column
  % (i.i. set flag!)

  if sum(ep_rem_col_flag==i)==0 & sum(kp_rem_col_flag==i)==0,

    keep_col_flag=[keep_col_flag i];

    keep_row_flag=[keep_row_flag i];

  end

end

% modified the rhs due to the known potentials

rhs=sparse(-sys_mat*(vs(1:k)')+full(rhs));

sys_mat=sys_mat(keep_row_flag,keep_col_flag); % prune the system matrix based
rhs=rhs(keep_row_flag,1);   % on the flags set above

```
% solve the system! EXPLOIT SPARSITY

v=full(sys_mat\rhs);
v_pntr=v_pntr(1,keep_row_flag)';
v_soln(v_pntr)=v;
% put back all the equi-potentials for the floating PEC/PMR objects
for i=1:num_pec_obj

  equal_pots=find(v_source==10000+3+i);

  [dum,num_eq_pots]=size(equal_pots);

  while num_eq_pots>1,

    v_soln(equal_pots)=v_soln(equal_pots(1))*ones(1,num_eq_pots);

    num_eq_pots=0;

  end

end
v_soln(known_pots)=v_source(known_pots);
% put the solution into a matrix form so that results can be displayed
v_source_mat=zeros(ycells+1,xcells+1);
for i=1:k
  pntr=find(ltable==i);
  v_source_mat(pntr)=v_soln(ltable(pntr));
end

% fix the corners, since they were not included in the system matrix

v_corner=insidepp*v_soln(xcells*4-3:xcells*4-3+ ...
                (xcells-2)*4-1)';


v_source_mat(1,1)=v_corner(1);
v_source_mat(1,xcells+1)=v_corner(2);
v_source_mat(ycells+1,xcells+1)=v_corner(3);
v_source_mat(ycells+1,1)=v_corner(4);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% end of matsolvf.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                              %
%% q_calc.m: Function to perform a flux integral of the D-Field through the  %
%%      rectangular "surface" shown below.                    %
%%                                              %
%% USAGE: [Q]=q_calc(V,er,dx,dy)   where: V is the Electric potential      %
%%                        er is relative permittivity       %
%%      o<---(x1,y1)---------------o  dx and dy are the grid spacing     %
%%      |_|_|_|_|_|_|_|_|_|_|_|_|_|_|                 %
%%      |_|_|_|_|Charge_|_|_|_|_|_|_|  V: (MxN) matrix       Units: V   %
%%      |_|_|_|_enclosed|_|_|_|_|_|  er: (M-1)x(N-1) matrix  Units: none %
%%      |||||||||||||||  dx and dy: scalar     Units: m   %
%%      o---------------(x2,y2)--->o  Q: scalar  Units: C/m (z-invariant) %
%%                        C (rotational sym)%
%% When the function is called, the user will use the crosshairs to indicate  %
%% the positions of two opposite corners.  The function will return the      %
%% amount of charge enclosed or 'error' if the user has entered the corners  %
%% outside of the region where V exists or the closed surface contains no     %
%% volume. (i.e. the two corners do not specify a box)               %
%%                                              %
%% This program has been modified to accomodate rotational symmetry        %
%% dpw 940430                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [q]=q_calc(v,e,dx,dy);
load mouse.emg -mat;
global cyl_flag
disp('')
disp('1. Press the left button for one corner of the box')
disp('2. Press the left button again for the other corner')
disp('3. Press the right button when you are done')
disp('')
disp('If you press the right button immediately after choosing this option')
disp('it will take you back to the previous screen.')
disp('')
%*******************
disp('Press any key to continue...')
%*******************
pause

[xx,yy,button]=ginput(1);
```

```
b1flag=0;
b2flag=0;
Lower=0;

while button~=Right_Button | b1flag==0 | b2flag==0,

  if button==Left_Button & Lower==0,
    b1flag=1;
    x(1)=xx;
    y(1)=yy;
    Lower=1;
  elseif button==Left_Button & Lower==1,
    b2flag=1;
    x(2)=xx;
    y(2)=yy;
    Lower=0;
  end

  [xx,yy,button]=ginput(1);

end

x=round(x/dx);
y=round(y/dy);
[r,c]=size(v);

if sum(x<0)==0 & sum(y<0)==0 & abs(x(1)-x(2))>1 & abs(y(1)-y(2))>1 & ...
     sum(x>c)==0 & sum(y>r)==0,

  q=0;
  eo=8.854e-12;


  if x(1)>x(2), x=[x(2) x(1)]; end
  if y(1)<y(2), y=[y(2) y(1)]; end

  toprow  = r-y(1); bottomrow = r-y(2)-1;
  leftcol = x(1)+1;    rightcol = x(2);

  for col=leftcol+1:rightcol

     elr_top   = 0.5*(e(toprow,col-1)   + e(toprow,col));
```

```
        elr_bottom = 0.5*(e(bottomrow,col-1) + e(bottomrow,col));
        if cyl_flag=='C'
          q = q + (elr_top    * (v(toprow+1,col)  - v(toprow,col)) + ...
              elr_bottom * (v(bottomrow,col) - v(bottomrow+1,col))) ...
                *(2*pi*(col-1/2)*dx);
        else
          q = q + elr_top    * (v(toprow+1,col)  - v(toprow,col)) + ...
              elr_bottom * (v(bottomrow,col) - v(bottomrow+1,col));
        end
    end
    for row=toprow+1:bottomrow
        etb_left  = 0.5*(e(row-1,leftcol)  + e(row,leftcol));
        etb_right = 0.5*(e(row-1,rightcol) + e(row,rightcol));

        if cyl_flag=='C'
          q = q + etb_left  * (v(row,leftcol+1) - v(row,leftcol)) ...
                    * (2*pi*(leftcol-1/2)*dy) + ...
            etb_right * (v(row,rightcol)  - v(row,rightcol+1)) ...
            * (2*pi*(rightcol-1/2)*dy);

        else
          q = q + etb_left  * (v(row,leftcol+1) - v(row,leftcol)) + ...
            etb_right * (v(row,rightcol)  - v(row,rightcol+1))
        end

    end

    q=q*eo;

    x(1)=(x(1)+0.5)*dx;
    x(2)=(x(2)-0.5)*dx;
    y(1)=(y(1)-0.5)*dy;
    y(2)=(y(2)+0.5)*dy;

    hold on
    plot([x(1) x(2) x(2) x(1) x(1)],[y(1) y(1) y(2) y(2) y(1)],'c1-')
    hold off
else
    q='error';
end


%%%%%%%%%%%%%%%%%%%%%% end of q_calc.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                                        %
%% solndom.m: This is used to start a new EM-Static problem.  It is chosen to %
%%         initialize a new solution domain.                %
%% updated 940101 dpw                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clc

newdomain='y';

if domain_flag==1,

  newdomain=input([bell,'WARNING: THIS WILL ERASE YOUR PREVIOUS', ...
                           ' DOMAIN, CONTINUE [n]?: '],'s');
  if strcmp(newdomain,[]), newdomain='n'; end

end

if newdomain=='y',

  domain_flag=1;

  clc
  disp('         DOMAIN REGION SCREEN        ')
  disp(' ')
  disp('The dimension entered here determines the area in which you will')
  disp('place your sources and media geometry.  This area is bordered by ')
  disp('the dashed blue line around the blue grid. ')
  disp('')

  rot_sym=input('Do you want to solve a rotationally symmetric system? [n] ','s');
  if strcmp(rot_sym,[]), rot_sym='R'; end
  if (rot_sym=='y')|(rot_sym=='Y')
    cyl_flag='C';
    if EM_flag=='E'
      enclosed_str='C';
    else
      enclosed_str=='A*m';
    end
  else
```

```matlab
        cyl_flag='R';

    end
xmax=-9999;
xmin=0;
while xmax@0,
    xmax=input(['Enter the dimension (in meters) of a', ...
                                  ' square solution domain: ']);
    if strcmp(xmax,[]), xmax=-1; end
    if xmax@0,
        disp([bell,'Only positive dimensions in this world, thank you!']);
    end
end
ymin=0;
ymax=xmax;
xcells=N_fine-2;
ycells=N_fine-2;
dxp=(xmax-xmin)/xcells;
dyp=(ymax-ymin)/ycells;
ymax=ymax+2*dyp;
xmax=xmax+2*dxp;
xcells=xcells+2;
ycells=ycells+2;
er_matrix=ones(ycells,xcells);
v_source=9999*ones((xcells+1)*(ycells+1),1);
v_source_coarse=9999*ones((N_coarse+1)*(N_coarse+1),1);
pec_pt_matrix=zeros(ycells+1,xcells+1);
pec_conn_matrix=zeros(2*ycells+1,2*xcells+1);
pcm=pec_conn_matrix;
dx=(xmax-xmin)/xcells;
dy=(ymax-ymin)/ycells;
charge_mat=zeros(ycells+1,xcells+1);
charge_mat_coarse=zeros(N_coarse+1,N_coarse+1);
cg_made=0;
geom_type='f';
redraw2
%*******************
    pause
%*******************
    hold off
end
%%%%%%%%%%%%%%%%%%%%%%%%% end of solndom.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

APPENDIX B
MATRIX METHOD TGT PROGRAMS

```
%  coefgenc.m
% This program will generate TGT coefficients using the
% matrix solution method for rotationally symmetric electrostatic systems.
% Functions required include: makemmc.m, makemlc.m and makemnc.m.
% Author: David P. Wells
% Date of last revision: 940116
M=input('Enter the Far Dirichlet Boundary Dimension ');
N=input('Enter the Computational Grid Dimension ');
M=M-2;
gammac=makemmc(M,N);
while M>N+2
    M=M-2;
    gammac=sparse(makemmc(M,N)-sparse(makemlc(M,N)*inv(gammac) ...
            *makemnc(M+2,N)));
end
TGT=(-1)*(inv(gammac)*makemnc(M,N));
TGT=[TGT;TGT(3*N+2,:)];        %% Adding left side TGT coefficients
TGT=[TGT(1,:);TGT];          %% based on symmetry.
TGT(4*N+4,1)=1;
for i=3*N-2:4*N-4
    TGT(i+7,i)=1;
end
TGT=full(TGT);
```

```
        function [x]=makemlc(M,N);
% This function makes the Ml matrix for generating TGT
% coefficients given the desired size of M (right side of layer)
% and N (computational grid dimension)
% for rotationally symmetric systems.
% David P. Wells 940302
l=N+2*M+2;
m=l-4;
top=N+(M-N)/2;
r=top-1/2;
side=M;
x=sparse(m,l);
row=1;
col=1;
for i=1:top
   x(row,col)=-1;
   row=row+1;
   col=col+1;
end
col=col+1;
row=row-1;
for i=1:M
   x(row,col)=-1*(1+1/(2*r));
   row=row+1;
   col=col+1;
end
col=col+1;
row=row-1;
for i=1:top
   x(row,col)=-1;
   row=row+1;
   col=col+1;
end
```

M1=full(makemlc(5,3))

M1 =

Columns 1 through 6

-1	0	0	0	0	0
0	-1	0	0	0	0
0	0	-1	0	0	0
0	0	0	-1	0	-8/7
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Columns 7 through 12

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
-8/7	0	0	0	0	0
0	-8/7	0	0	0	0
0	0	-8/7	0	0	0
0	0	0	-8/7	0	-1
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Columns 13 through 15

0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
-1	0	0
0	-1	0
0	0	-1

```
        function [Mm]=makemmc(M,N)
% This function makes layer self matrices (Mm) for rotationally
% symmetric systems given M (length of right side of layer)
% and N (computational grid dimension).
% David P. Wells 940115
m=N+2*M-2;
top=N+(M-N)/2;
side=M;
rtcorn=top-1+side;
e=ones(m,1);
Mm=spdiags([-1*e 4*e -1*e],-1:1,m,m);
r=1/2;
for i=1:top
   if i~=top
     Mm(i,i+1)=(-1)*(1+1/(2*r));
   end
   if i~=1
     Mm(i,i-1)=(-1)*(1-1/(2*r));
   end
r=r+1;
end
r=1/2;
for i=m:-1:rtcorn
   if i~=rtcorn
     Mm(i,i-1)=(-1)*(1+1/(2*r));
   end
   if i~=m
     Mm(i,i+1)=(-1)*(1-1/(2*r));
   end
r=r+1;
end
```

Mm=full(makemmc(5,3))

Mm =
Columns 1 through 6

4	-2	0	0	0	0
-2/3	4	-4/3	0	0	0
0	-4/5	4	-6/5	0	0
0	0	-6/7	4	-1	0
0	0	0	-1	4	-1
0	0	0	0	-1	4
0	0	0	0	0	-1
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Columns 7 through 11

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
-1	0	0	0	0
4	-1	0	0	0
-1	4	-6/7	0	0
0	-6/5	4	-4/5	0
0	0	-4/3	4	-2/3
0	0	0	-2	4

102

```
        function [x]=makemnc(M,N);
% This function makes the Mn matrix for generating TGT
% coefficients given the desired size of M (length of right side of layer)
% and N (computational grid size) for z-invariant systems.
% David P. Wells 940115
M=M-2;
m=N+2*M-2;
n=m-4;
top=N+(M-N)/2;
r=top+1/2;
side=M;
x=sparse(m,n);
row=1;
col=1;
for i=1:top
   x(row,col)=-1;
   row=row+1;
   col=col+1;
end
col=col-1;
row=row+1;
for i=1:M
   x(row,col)=-1*(1-1/(2*r));
   row=row+1;
   col=col+1;
end
col=col-1;
row=row+1;
for i=1:top
   x(row,col)=-1;
   row=row+1;
   col=col+1;
end
end
```

```
Mn=full(makemnc(5,3))

Mn =
Columns 1 through 6
   -1      0      0      0      0      0
    0     -1      0      0      0      0
    0      0     -1      0      0      0
    0      0      0      0      0      0
    0      0    -6/7     0      0      0
    0      0      0    -6/7     0      0
    0      0      0      0    -6/7     0
    0      0      0      0      0      0
    0      0      0      0     -1      0
    0      0      0      0      0     -1
    0      0      0      0      0      0

Column 7
    0
    0
    0
    0
    0
    0
    0
    0
    0
    0
   -1
```

```
% coefmagc.m
% This program will generate TGT coefficients using the
% matrix solution method for magnetostatic
% rotationally symmetric systems.
% Functions required include: magmmc.m, magmlc.m and magmnc.m.
% Author: David P. Wells
% Date of last revision: 940514
M=input('Enter the Far Dirichlet Boundary Dimension ');
N=input('Enter the Computational Grid Dimension ');
M=M-2;
gammac=magmmc(M,N);
while M>N+2
    M=M-2;

gammac=sparse(magmmc(M,N)-sparse(magmlc(M,N)*inv(gammac)*magmnc(M+2,N)));
end
TGT=(-1)*(inv(gammac)*makemnc(M,N));
[leng,width]=size(TGT);
TGT=[TGT;zeros(1,width)];       %% Adding left side TGT coefficients
TGT=[zeros(1,width);TGT];       %% based on symmetry.
TGT(4*N+4,1)=0;                 %% centerline has zero potential
for i=3*N-2:4*N-4
    TGT(i+7,i)=0;
end
TGT=full(TGT);
```

```
        function [x]=magmlc(M,N);
% This function makes the Ml matrix for generating TGT
% coefficients given the desired size of M (right side of layer)
% and N (computational grid dimension)
% for magnetostatic rotationally symmetric systems.
% David P. Wells 940514
l=N+2*M+2;
m=l-4;
top=N+(M-N)/2;
r=top;
side=M;
x=sparse(m,l);
row=1;
col=1;
for i=1:top
   x(row,col)=-1;
   row=row+1;
   col=col+1;
end
col=col+1;
row=row-1;
for i=1:M
   x(row,col)=-1*(1+1/(2*r+1));
   row=row+1;
   col=col+1;
end
col=col+1;
row=row-1;
for i=1:top
   x(row,col)=-1;
   row=row+1;
   col=col+1;
end
```

Ml=full(magmlc(5,3))

Ml =

Columns 1 through 6

-1	0	0	0	0	0
0	-1	0	0	0	0
0	0	-1	0	0	0
0	0	0	-1	0	-10/9
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Columns 7 through 12

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
-10/9	0	0	0	0	0
0	-10/9	0	0	0	0
0	0	-10/9	0	0	0
0	0	0	-10/9	0	-1
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Columns 13 through 15

0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
-1	0	0
0	-1	0
0	0	-1

```
        function [Mm]=magmmc(M,N)
% This function makes layer self matrices (Mm) for rotationally
% symmetric magnetostatic systems
% given M (length of right side of layer)
% and N (computational grid dimension).
% David P. Wells 950415
m=N+2*M-2;
top=N+(M-N)/2;
side=M;
rtcorn=top-1+side;
e=ones(m,1);
Mm=spdiags([-1*e 4*e -1*e],-1:1,m,m);
r=1;
for i=1:top
   Mm(i,i)=Mm(i,i)-(1/(2*r+1))+(1/(2*r-1));
   if i~=top
     Mm(i,i+1)=(-1)*(1+1/(2*r+1));
   end
   if i~=1
     Mm(i,i-1)=(-1)*(1-1/(2*r-1));
   end
   r=r+1;
end
r=r-1;
for i=(top+1):(rtcorn-1)
   Mm(i,i)=Mm(i,i)-(1/(2*r+1))+(1/(2*r-1));
end
r=1;
for i=m:-1:rtcorn
   Mm(i,i)=Mm(i,i)-(1/(2*r+1))+(1/(2*r-1));
   if i~=rtcorn
     Mm(i,i-1)=(-1)*(1+1/(2*r+1));
   end
   if i~=m
     Mm(i,i+1)=(-1)*(1-1/(2*r-1));
   end
   r=r+1;
end
```

Mm=full(magmmc(5,3))

Mm =

Columns 1 through 6

14/3	-4/3	0	0	0	0
-2/3	62/15	-6/5	0	0	0
0	-4/5	142/35	-8/7	0	0
0	0	-6/7	254/63	-1	0
0	0	0	-1	254/63	-1
0	0	0	0	-1	254/63
0	0	0	0	0	-1
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Columns 7 through 11

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
-1	0	0	0	0
254/63	-1	0	0	0
-1	254/63	-6/7	0	0
0	-8/7	142/35	-4/5	0
0	0	-6/5	62/15	-2/3
0	0	0	-4/3	14/3

```
        function [x]=magmnc(M,N);
% This function makes the Mn matrix for generating magnetostatic TGT
% coefficients given the desired size of M (length of right side of layer)
% and N (computational grid size) for rotationally symmetric systems.
% David P. Wells 940514
M=M-2;
m=N+2*M-2;
n=m-4;
top=N+(M-N)/2;
r=top+1;
side=M;
x=sparse(m,n);
row=1;
col=1;
for i=1:top
   x(row,col)=-1;
   row=row+1;
   col=col+1;
end
col=col-1;
row=row+1;
for i=1:M
   x(row,col)=-1*(1-1/(2*r-1));
   row=row+1;
   col=col+1;
end
col=col-1;
row=row+1;
for i=1:top
   x(row,col)=-1;
   row=row+1;
   col=col+1;
end
end
```

Mn=full(magmnc(5,3))

Mn =

Columns 1 through 6

-1	0	0	0	0	0
0	-1	0	0	0	0
0	0	-1	0	0	0
0	0	0	0	0	0
0	0	-6/7	0	0	0
0	0	0	-6/7	0	0
0	0	0	0	-6/7	0
0	0	0	0	0	0
0	0	0	0	-1	0
0	0	0	0	0	-1
0	0	0	0	0	0

Column 7

0
0
0
0
0
0
0
0
0
0
-1

```
% coefgen.m
% This program will generate TGT coefficients using the
% matrix solution method for z-invariant systems.
% Functions required include: makemm.m, makeml.m and makemn.m.
% Author: David P. Wells
% Date of last revision: 940116
M=input('Enter the Dirichlet Boundary Dimension ');
N=input('Enter the Computational Grid Dimension ');
M=M-2;
gamma=makemm(M);
while M>N+2
    M=M-2;
    gamma=sparse(makemm(M)-sparse(makeml(M)*inv(gamma)*makemn(M+2)));
end
TGT=(-1)*full(inv(gamma)*makemn(M));
```

```
        function [x]=makeml(M);
% This function makes the Ml matrix for generating TGT
% coefficients given the desired size of M (layer dimension).
% It is a modified version of makemn since pattern is same but transposed.
% David P. Wells 940115
M=M+2;
m=4*M-4;
n=m-8;
N=M-2;
x=sparse(m,n);
x(m,1)=-1;
row=1;
col=1;
while row<(m-(N+1))
    row=row+1;
    for i=1:N
        x(row,col)=-1;
        row=row+1;
        col=col+1;
    end
col=col-1;
end
row=row+1;
  for i=1:(N-1)
      x(row,col)=-1;
  row=row+1;
  col=col+1;
end
x=x';
```

M1=full(makeml(5))

M1 =
Columns 1 through 6

```
     0    -1     0     0     0     0
     0     0    -1     0     0     0
     0     0     0    -1     0     0
     0     0     0     0    -1     0
     0     0     0     0     0    -1
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
```

Columns 7 through 12

```
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0    -1     0     0     0     0
     0     0    -1     0     0     0
     0     0     0    -1     0     0
     0     0     0     0    -1     0
     0     0     0     0     0    -1
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
     0     0     0     0     0     0
```

Columns 13 through 18

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	-1	0	0	0	0
0	0	-1	0	0	0
0	0	0	-1	0	0
0	0	0	0	-1	0
0	0	0	0	0	-1
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Columns 19 through 24

0	0	0	0	0	-1
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	-1	0	0	0	0
0	0	-1	0	0	0
0	0	0	-1	0	0
0	0	0	0	-1	0

```
        function [Mm]=makemm(M)
% This function makes layer self matrices (Mm) for z-invariant systems
% given M (layer size).
% David P. Wells 940115
m=4*M-4;
e=ones(m,1);
Mm=spdiags([-1*e 4*e -1*e],-1:1,m,m);
Mm(1,m)=-1,
Mm(m,1)=-1;
```

Mm=full(makemm(5))

Mm =
 Columns 1 through 6

4	-1	0	0	0	0
-1	4	-1	0	0	0
0	-1	4	-1	0	0
0	0	-1	4	-1	0
0	0	0	-1	4	-1
0	0	0	0	-1	4
0	0	0	0	0	-1
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
-1	0	0	0	0	0

 Columns 7 through 12

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
-1	0	0	0	0	0
4	-1	0	0	0	0
-1	4	-1	0	0	0
0	-1	4	-1	0	0
0	0	-1	4	-1	0
0	0	0	-1	4	-1
0	0	0	0	-1	4
0	0	0	0	0	-1
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Columns 13 through 16

0	0	0	-1
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
-1	0	0	0
4	-1	0	0
-1	4	-1	0
0	-1	4	-1
0	0	-1	4

```
        function [x]=makemn(M);
% This function makes the Mn matrix for generating TGT
% coefficients given the desired size of M (layer size).
% David P. Wells 940115
m=4*M-4;
n=m-8;
N=M-2;
x=sparse(m,n);
x(m,1)=-1;
row=1;
col=1;
while row<(m-(N+1))
   row=row+1;
   for i=1:N
      x(row,col)=-1;
      row=row+1;
      col=col+1;
   end
col=col-1;
end
row=row+1;
  for i=1:(N-1)
     x(row,col)=-1;
  row=row+1;
  col=col+1;
end
```

Mn=full(makemn(5))

Mn =
Columns 1 through 6

```
 0    0    0    0    0    0
-1    0    0    0    0    0
 0   -1    0    0    0    0
 0    0   -1    0    0    0
 0    0    0    0    0    0
 0    0   -1    0    0    0
 0    0    0   -1    0    0
 0    0    0    0   -1    0
 0    0    0    0    0    0
 0    0    0    0   -1    0
 0    0    0    0    0   -1
 0    0    0    0    0    0
 0    0    0    0    0    0
 0    0    0    0    0    0
 0    0    0    0    0    0
-1    0    0    0    0    0
```

Columns 7 through 8

```
 0    0
 0    0
 0    0
 0    0
 0    0
 0    0
 0    0
 0    0
 0    0
 0    0
 0    0
 0    0
-1    0
 0    0
-1    0
 0   -1
 0    0
```

APPENDIX C
MONTE CARLO METHOD TGT PROGRAMS

```
% cylcoffs.m
% Purpose is to create matrix of coefficients for rotationally symmetric
% system for use in EMAG via the Monte Carlo Method.
% Required functions include: fncylwlk.m and unravel.m
% TGT matrix will be saved in a file named cymatrix.mat with variable name
% "coeffs"
% David P. Wells 931231
N=input('Enter Computational Grid Dimension ');
M=input('Enter Dirichlet Boundary Size (odd/even if comp grid odd/even)');
R=input('Enter the Number of Walkers ');
coeffs=[];
for i=2:N+2                          %% top
    new_coeffs=fncylwlk(N,M,R,1,(M-N)/2,i);
    coeffs=[coeffs;new_coeffs];
end
for i=1:N+1                          %% right side
    new_coeffs=fncylwlk(N,M,R,1,(M-N)/2+i,N+2);
    coeffs=[coeffs;new_coeffs];
end
for i=N+1:-1:2                       %% bottom
    new_coeffs=fncylwlk(N,M,R,1,(M+N)/2+1,i);
    coeffs=[coeffs;new_coeffs];
end
coeffs=[coeffs(1,:);coeffs];
coeffs=[coeffs;coeffs(3*N+3,:)];
counter=0;
for i=(M+N)/2:-1:((M-N+4)/2)                %% left side using symmetry
    new_coeffs=zeros(1,4*N-4);
    new_coeffs(3*N-3+counter+1)=1;
    counter=counter+1;
    coeffs=[coeffs;new_coeffs];
end
new_coeffs=zeros(1,4*N-4);
new_coeffs(1)=1;
coeffs=[coeffs;new_coeffs];
save cymatrix coeffs
```

```
        function [coeffs]=fncylwlk(N,M,W,loops,startrow,startcol)
%% David P. Wells Updated 931027. This function returns a row of the TGT matrix for
% rotationally symmetric systems using the MCM given the coordinates
% of the walker's starting point and the following input parameters:
% N= inner matrix dimension
% M= outer matrix dimension
% loops= number of loops
% W= # of walkers per loop
% startrow= walker release row
% startcol= release column
L=2;
T=(M-N+2)/2;
R=N+1;
B=(M+N)/2;
compgrid=ze os(N,N);
crnrow=T-1;    %corners of inner grid
crncol=L-1;
for i=1:loops
   inbound=zeros(1,W);
   outbound=zeros(1,W);
   row=ones(1,W);
   col=ones(1,W);
   row=startrow*row;
   col=startcol*col;
   while length(row)~=0
      dr=rand(1,length(row));
      up=(dr<=.25);
      dn=((dr>.25)&(dr<=.5));
      comprand=.25*(3-ones(1,length(col))./(2*(col-1.5)));
      lt=((dr>.5)&(dr<=comprand));
      rt=(dr>comprand);
      row=row+up-dn;
      col=col+rt-lt;
      inbound=((row==T)&((col>=L)&(col<=R)));
      inbound=(inbound|((row==B)&((col>=L)&(col<=R))));
      inbound=(inbound|((col==R)&((row>=T)&(row<=B))));
      inbound=(inbound|((col==L)&((row>=T)&(row<=B))));
      rowin=row(find(inbound==1));
      colin=col(find(inbound==1));
      for k=1:length(rowin)
      compgrid(rowin(k)-crnrow,colin(k)-crncol)= ...
                        compgrid(rowin(k)-crnrow,colin(k)-crncol)+1;
   end
```

122

```
        outbound=((row==1)|(row==M)|(col==1)|(col==(M+N+2)/2));
        row(find((outbound==1)|(inbound==1)))=[];
        col(find((outbound==1)|(inbound==1)))=[];
     end
end
coeffs=unravel(compgrid);
coeffs=(1/(loops*W))*coeffs;
```

```
        function x=unravel(y)
%function unravel takes boundary coefficients from around compgrid and puts
%them in a row vector using a spiral numbering scheme.
% David P. Wells 931030
D=size(y);
d=D(1);
x=y(1,:);
x=[x (y(2:d,d))'];
for i=d-1:-1:1
  x=[x y(d,i)];
end
for i=d-1:-1:2
  x=[x y(i,1)];
end
```

```
% Dave Wells Updated:931026
% rctcoffs.m
% Purpose is to create a TGT matrix of coefficients for rectangular
% coordinate systems for use in EMAG
% Coefficients will be saved in a file named rcmatrix.mat
% with variable name coeffs
% The coefficient pattern is as follows:
%
%        A B C D E
%        P 1 2 3 F
%        O 8   4 G
%        N 7 6 5 H
%        M L K J I
%
% Here, the letters represent the relative location
% on the boundary layer and the numbers represent the relative
% locations on the computational grid edges.
% Row A, Column 1 of the coefficient matrix named "coeffs" is the
% coefficient linking point A on the boundary to point #1 on
% the computational grid. For this example, "coeffs" would be a
% 16x8 matrix.
% This program requires fnrctwlk.m and unravel.m to operate.
N=input('enter the inner grid dimension ');
M=input('enter outer grid dimension (odd if inner dimension odd, even if inner is even) ');
R=input('enter the number of walkers ');
coeffs=[];
for i=(M-N)/2:(M+N+2)/2             %  top row
   new_coeffs=fnrctwlk(N,M,R,1,(M-N)/2,i);
   coeffs=[coeffs;new_coeffs];
end
for i=(M-N+2)/2:(M+N+2)/2           % right side
   new_coeffs=fnrctwlk(N,M,R,1,i,(M+N+2)/2);
   coeffs=[coeffs;new_coeffs];
end
for i=(M+N)/2:-1:(M-N)/2            % bottom row
   new_coeffs=fnrctwlk(N,M,R,1,(M+N+2)/2,i);
   coeffs=[coeffs;new_coeffs];
end
for i=(M+N)/2:-1:(M-N+2)/2
   new_coeffs=fnrctwlk(N,M,R,1,i,(M-N)/2); % left side
   coeffs=[coeffs;new_coeffs];
end
save rcmatrix coeffs
```

```
        function[coeffs]=fnrctwlk(N,M,W,loops,startrow,startcol)
%% David P. Wells Updated 931027
%  This function returns a row of the TGT matrix for z-invariant systems
%  given the coordinates of the walker's starting point and the following
%  input parameters:
%  N= inner matrix dimension
%  M= outer matrix dimension
%  W= number of walkers per loop
%  loops= number of loops (use one unless running out of memory)
L=(M-N)/2+1;    % defining left side of computational grid
T=L;            % defining top
R=(M+N)/2;      % defining right
B=R;            % defining bottom
pr=.25;         % fraction moving right
pl=.5;          % fraction moving left
pu=.75;         % fraction moving up
compgrid=zeros(N,N);    % initializing computational grid location matrix
crnrow=T-1;     %corners of inner grid
crncol=L-1;


for i=1:loops     % loops used only to avoid "out of memory" situation
    inbound=zeros(1,W);     % initializing collision with inner grid matrix
    outbound=zeros(1,W);    % initializing collision with outer grid matrix
    row=ones(1,W);          % creating walker row position vector
    col=ones(1,W);          % creating walker column position vector
    row=startrow*row;       % initializing starting release point
    col=startcol*col;       %
    while length(row)~=0    % looping until all walkers have come to rest on
                    % inner or outer bound
      dr=rand(1,length(row));   % creating walker direction vector
      rt=(dr<=.25);             % choosing those to move right
      lt=((dr>.25)&(dr<=.5));   % choosing those to move left
      up=((dr>.5)&(dr<=.75));   % choosing those to move up
      dn=(dr>.75);              % choosing those to move down
      row=row+up-dn;            % moving some left and some right
      col=col+rt-lt;            % moving some up and some down
      inbound=((row==T)&((col>=L)&(col<=R)));   % check for inside collisions
      inbound=(inbound|((row==B)&((col>=L)&(col<=R))));  % ditto
      inbound=(inbound|((col==R)&((row>=T)&(row<=B))));  % ditto
      inbound=(inbound|((col==L)&((row>=T)&(row<=B))));  % ditto
      rowin=row(find(inbound==1));  % isolating walkers on inner boundary
      colin=col(find(inbound==1));  % ditto
      for k=1:length(rowin)         % record loc of walkers on inner bound
```

```
        compgrid(rowin(k)-cmrow,colin(k)-cmcol)= ...
                        compgrid(rowin(k)-cmrow,colin(k)-cmcol)+1;
    end
    outbound=((row==1)|(row==M)|(col==1)|(col==M)); % finding walkers on
                        % outer boundary
    row(find((outbound==1)|(inbound==1)))=[]; % discarding walkers
                        % on inner or outer bounds
    col(find((outbound==1)|(inbound==1)))=[]; % ditto
  end
end
coeffs=unravel(compgrid); % organizing coeffs using spiral number scheme
coeffs=(1/(loops*W))*coeffs; % normalizing coefficients to # walkers released
```

REFERENCES

1. Manke, Jr., R. P., *EMAG a 2-D Electrostatic and Magnetostatic Solver in MATLAB*, Master's Thesis, Rose-Hulman Institute of Technology, Terre Haute, IN, October 1992.

2. Wells, D. P. and Lebaric, J. E., "EMAG 2.0 - Enhanced 2D Electrostatic and Magnetostatic Solver in MATLAB," *Conference Proceedings of the 10th Annual Review of Progress in Applied Computational Electromagnetics*, Monterey, CA, 25 March 1992.

3. Sadiku, M. N. O., "Monte Carlo Methods in an Introductory Electromagnetic Course," *IEEE Transactions on Education*, Vol 33. No. 1, February 1990.

4. Lebaric, J. E., "Open Boundary Simulation for FD's - Transparent Grid Termination," paper presented to Computational Electromagnetics class at Naval Postgraduate School, Monterey, CA, November 1993.

5. Haykin, S. S., *An Introduction to Analog and Digital Communications*, John Wiley & Sons, Inc., New York, NY, 1989.

6. Cheng, D. K., *Field and Wave Electromagnetics*, Addison -Wesley Publishing Co., Reading, MA, 1989.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center   2
    Cameron Station
    Alexandria, VA 22304-6145

2. Library, Code 52   1
    Naval Postgraduate School
    Monterey, CA 93943-5101

3. Chairman, Code EC   1
    Department of Electrical and Computer Engineering
    Naval Postgraduate School
    Monterey, CA 93943-5121

4. Dr. David C. Jenn, Code EC/Jn   1
    Naval Postgraduate School
    Monterey, CA 93943-5121

5. Director, Training and Education   1
    MCCDC, Code C46
    1019 Elliot Road
    Quantico, VA 22143-5027

6. Dr. Jovan E. Lebaric   1
    Campus Box 119
    Rose-Hulman Institute of Technology
    5500 Wabash Ave.
    Terre Haute, IN 47803

7. Capt. David P. Wells, USMC   2
    1074 Minerva Ave.
    Columbus, OH 43229